Technische Universität Braunschweig

ISF

Custom-Tailored Product Line Extraction

Von der

Carl-Friedrich-Gauß-Fakultät

der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines

Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von

David Wille

geboren am 28.04.1989

in Münster

Eingereicht am:	31.08.2018
Disputation am:	30.11.2018
1. Referentin:	Prof. Dr.-Ing. Ina Schaefer
2. Referent:	Prof. Dr. rer. nat. Ralf H. Reussner

2019

Unser Wille ist nur der Wind, der uns drängt und dreht;
weil wir selber die Sehnsucht sind, die in Blüten steht.

Rainer Maria Rilke

Abstract

Industry faces an increasing number of challenges regarding the functionality, efficiency and reliability of developed software systems. Especially in domains with safety-critical systems, such as the automotive domain or the aviation domain, these challenges increase the complexity and costs during development. A common approach to reduce the complexity and involved development effort are model-based languages, such as MATLAB/SIMULINK and statecharts. Starting on a high level of abstraction, these languages allow to refine the functionality to be developed step-by-step.

While this approach helps companies during development of single systems, the customers' high demand for software that is specifically configured towards their requirements is an increasing challenge. As a result, companies have to efficiently develop variants that have slightly varying functionality, but still are highly similar. As reimplementation of complex functionality for each variant is no option, a common solution to cope with this problem is copying existing solutions and modifying them to changed needs. In the short-run, this so-called clone-and-own approach allows to save development costs and enables developers to easily reuse existing solutions by adding or removing parts and modifying existing functionality for a specific customer. However, this approach also involves high risks for future maintenance as the relations between the copied systems are rarely documented. For example, fixing bugs becomes a tedious task as the systems have to be maintained in isolation, and developers have to manually identify and fix them in all variants. Thus, with a growing number of such potentially large system copies, the resulting effort can become a limiting factor for a company's productivity as increasing capacity is spent on maintaining existing systems.

For a more structured reuse strategy, academia promotes to apply software product lines. This approach allows to develop and automatically generate variants from a set of reusable artifacts constituting their common and varying parts. While the approach was successfully adopted by many companies, its adoption bares initial risks as large manual effort is required to identify the common and varying parts of existing variants and to encode them in a future software product line.

To overcome these problems, this thesis contributes a variability mining algorithm for existing model variants to allow their semi-automatic migration to a software product line. As industry uses a variety of different modeling languages, the focus of the approach lies on an easy adaptation for different languages. Furthermore, the approach can be custom-tailored to include domain knowledge or language-specific details in the mining process and, thus, to identify relations expected by domain experts. The first step of the approach performs a high-level analysis of variants to identify outliers (e.g., variants that diverged too much from the rest) and clusters of variants relevant for a detailed variability analysis. The second step executes variability mining to identify low-level variability relations for these clusters supporting developers during maintenance by showing common and varying parts of the corresponding variants. The third step uses these detailed variability relations to allow automatic migration of the compared variants to a delta-oriented software product line. The proposed approach is evaluated using publicly available case studies with industrial background as well as model variants provided by an industry partner.

Zusammenfassung

Die Industrie steht einer steigenden Anzahl an Herausforderungen bezüglich der Funktionalität, Effizienz und Zuverlässigkeit von entwickelten Softwaresystemen gegenüber. Speziell in Bereichen mit sicherheitsrelevanten Systemen, wie der Automobil- oder Flugzeugindustrie, führen diese Herausforderungen zu einer Komplexitäts- und Kostensteigerung während der Entwicklung. Um diese Komplexität und den verbundenen Entwicklungsaufwand zu reduzieren, werden häufig modellbasierte Sprachen wie MATLAB/SIMULINK oder Zustandsautomaten eingesetzt. Beginnend auf einem hohen Abstraktionslevel ermöglichen diese Sprachen, die entwickelte Funktionalität Schritt für Schritt zu verfeinern.

Obwohl diese Herangehensweise die Unternehmen während der Entwicklung von Einzelsystemen unterstützt, führt die große Nachfrage nach speziell auf Kundenwünsche zugeschnittener Software zu neuen Herausforderungen. Entsprechend müssen Unternehmen effizient Varianten entwickeln, die geringfügig unterschiedliche Funktionalität aufweisen, aber grundsätzlich sehr ähnlich sind. Da eine Neuimplementierung von komplexer Funktionalität für jede dieser Varianten keine Option darstellt, kopieren Unternehmen häufig existierende Lösungen und passen sie auf veränderte Bedürfnisse an. Auf kurze Sicht ermöglicht dieser sogenannte clone-and-own-Ansatz Entwicklungskosten zu sparen, da er Entwicklern einfach erlaubt, durch Hinzufügen, Entfernen oder Modifizieren von Funktionalität die existierenden Lösungen für einzelne Kunden wiederzuverwenden. Jedoch birgt der Ansatz große Risiken für die zukünftige Wartung, da die Beziehungen zwischen den kopierten Systemen in den seltensten Fällen dokumentiert werden. Zum Beispiel wird die Behebung von Fehlern zu einer langwierigen Aufgabe, da die Systeme einzeln gepflegt werden müssen und Entwickler gezwungen sind, die betroffenen Teile manuell in jeder der Varianten zu identifizieren und zu reparieren. Somit kann mit einer wachsenden Anzahl an möglicherweise umfangreichen Systemkopien der entsprechende Aufwand zu einem limitierenden Faktor für die Produktivität eines Unternehmens werden, da zunehmend Kapazität für die Wartung der Systeme benötigt wird.

Als eine Strategie für strukturiertere Wiederverwendung werden Softwareproduktlinien in der Forschung vorgeschlagen. Dieser Ansatz ermöglicht die Entwicklung und automatische Generierung von Varianten aus einer Menge von wiederverwendbaren Artefakten für die gemeinsamen und unterschiedlichen Softwareteile. Obwohl dieser Ansatz von vielen Unternehmen erfolgreich übernommen wurde, birgt die Einführung der Techniken anfängliche Risiken, da viel manueller Aufwand nötig ist, um die gemeinsamen und unterschiedlichen Teile der Varianten zu identifizieren und in eine Softwareproduktlinie zu überführen.

Um dieses Problem zu lösen, bietet diese Arbeit einen Variabilitätsidentifikationsalgorithmus, der existierende Modellvarianten semi-automatisch in eine Softwareproduktlinie überführt. Da in der Industrie eine große Menge an unterschiedlichen Modellierungssprachen eingesetzt wird, liegt der Fokus auf einer einfachen Adaption für verschiedene Sprachen. Zusätzlich ermöglicht es der Ansatz, durch Einbeziehung von Expertenwissen oder sprachspezifische Details die Variabilitäts-

identifikation auf bestimmte Gegebenheiten anzupassen, sodass die von Experten erwarteten Relationen identifiziert werden. Der erste Schritt des Ansatzes führt eine Analyse der Varianten auf einem hohen Abstraktionslevel durch, um Außenseiter (z.B. Varianten die sich zu weit vom Rest entwickelt haben) und Cluster von für die Vergleiche relevanten Varianten zu identifizieren. Der zweite Schritt führt die Variabilitätsidentifikation durch, um Variabilitätsrelationen auf niedrigem Abstraktionslevel für diese Cluster zu identifizieren und mit den aufgezeigten Gemeinsamkeiten und Unterschieden die Entwickler bei der Wartung der Varianten zu unterstützen. Der dritte Schritt nutzt diese detaillierten Variabilitätsrelationen für eine automatische Migration der verglichenen Varianten in eine delta-orientierte Softwareproduktlinie. Der gezeigte Ansatz ist an Fallstudien mit industriellem Kontext sowie Modellvarianten eines Industriepartners evaluiert worden.

Acknowledgements

Many people influenced and supported me over the years to shape the person I am today. In the following, I want to sincerely thank the people most related with this thesis.

First and foremost, I thank my supervisor Ina Schaefer for giving me the opportunity to do my PhD at her institute. She approached me during one of her lectures, which I attended as a Bachelor student, and offered me a job as a research assistant. Over the following two years my growing interest in research together with her continued encouragement and support gave rise to the wish of continuing my work on variability mining. During my time as a PhD student, she offered me many opportunities to gain personal as well as professional experiences and left me a lot of freedom to follow my research interests while supporting me when necessary. Furthermore, I thank my reviewer Ralph Reussner for taking the effort of reading this thesis with all linked duties.

In addition, I want to sincerely thank all other people who collaborated with me during my PhD and supported me in many different ways. Most of all, I want to thank Christoph Seidl for his continued interest in my work, our long discussions and his great advice not only on work related topics, but also as a dear friend on life after PhD and personal life in general. His broad knowledge and ability to point out fine details that make the difference between great and awesome ideas gave many of my solutions the final touch. I also want to thank Sandro Schulze for his valuable comments on many papers and his creativity when discussing future research directions. There are not enough lives to pursue all of them! Furthermore, I want to thank Kenny Wehling from Volkswagen AG as well as Önder Babur and Loek Cleophas from TU Eindhoven for our great collaboration.

I thank my colleagues from TU Braunschweig for the great team spirit. It was always a pleasure working with you, visiting conferences together and having fun during lunch break. The "class trip" to GPCE 2016 in Amsterdam will never be forgotten! Over the years, I received support by many students, which I supervised. Especially, I want to thank Tobias Runge and Alexander Schlie, who helped to implement and evaluate many of my ideas and who are PhD students themselves now. I also thank Richard Jockusch for his initial work on the hierarchy shift and dispersion detection.

Many thanks goes to my friends from Oyten and Braunschweig, who made these years enjoyable. Special thanks goes to my best friend Nicolas Wolff, whom I have known since first grade. Our Wednesday burger ritual helped me to get my mind off work and, in turn, to find some of the most fruitful solutions for problems that I was stuck with.

I thank my parents for an untroubled childhood with many opportunities to visit foreign countries, making me the open-minded person I am today. Thank you for fostering my education by letting me follow my interests without worrying about financing them. Special thanks goes to my mother Christa Wille for always lending me an ear and giving me great advice when I needed it.

Finally, I thank my beloved girlfriend Dorothee Ottow, who has been a great support in many difficult situations and always tried to sheer me up when I wanted to give up. Thank you for your empathy and giving me the strength to continue my work. Your relaxed attitude made me realize that a little bit of chaos is not the end of the world and helped to confine my perfectionism.

Publications

This doctoral thesis is based on the following peer-reviewed publications ordered by relevance, starting with the most relevant.

[1] **D. Wille**, Ö. Babur, L. Cleophas, C. Seidl, M. van den Brand, and I. Schaefer. "Improving Custom-Tailored Variability Mining Using Outlier and Cluster Detection". In: *Science of Computer Programming* 163 (2018), pp. 62–84.

[2] **D. Wille**, S. Schulze, C. Seidl, and I. Schaefer. "Custom-Tailored Variability Mining for Block-Based Languages". In: *Proc. of the Intl. Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE, 2016, pp. 271–282.

[3] **D. Wille**, T. Runge, C. Seidl, and S. Schulze. "Extractive Software Product Line Engineering Using Model-Based Delta Module Generation". In: *Proc. of the Intl. Workshop on Variability Modeling in Software-Intensive Systems (VaMoS)*. ACM, 2017, pp. 36–43.

[4] C. Seidl, **D. Wille**, and I. Schaefer. "Software Reuse – From Cloned Variants to Managed Software Product Lines". In: *Book on Automotive Software Engineering*. to be published. Atlantis Press/Springer, 2019.

[5] A. Schlie, **D. Wille**, S. Schulze, L. Cleophas, and I. Schaefer. "Detecting Variability in MATLAB/Simulink Models: An Industry-Inspired Technique and its Evaluation". In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*. ACM, 2017, pp. 215–224.

[6] **D. Wille**, S. Schulze, and I. Schaefer. "Variability Mining of State Charts". In: *Proc. of the Intl. Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2016, pp. 63–73.

[7] **D. Wille**, S. Holthusen, S. Schulze, and I. Schaefer. "Interface Variability in Family Model Mining". In: *Proc. of the Intl. Workshop on Model-Driven Approaches in Software Product Line Engineering (MAPLE)*. ACM, 2013, pp. 44–51.

[8] **D. Wille**. "Managing Lots of Models: The FaMine Approach". In: *Proc. of the Intl. Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 817–819.

[9] S. Holthusen, **D. Wille**, C. Legat, S. Beddig, I. Schaefer, and B. Vogel-Heuser. "Family Model Mining for Function Block Diagrams in Automation Software". In: *Proc. of the Intl. Workshop on Reverse Variability Engineering (REVE)*. ACM, 2014, pp. 36–43.

Further peer-reviewed publications directly related to this thesis.

[10] **D. Wille**, K. Wehling, C. Seidl, M. Pluchator, and I. Schaefer. "Variability Mining of Technical Architectures". In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*. Vol. A. **HITACHI Young Best Paper Award – Research Track**. ACM, 2017, pp. 39–48.

[11] **D. Wille**, M. Tiede, S. Schulze, C. Seidl, and I. Schaefer. "Identifying Variability in Object-Oriented Code Using Model-Based Code Mining". In: *Proc. of the Intl. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Vol. 9953. Lecture Notes in Computer Science. Springer, 2016, pp. 547–562.

[12] A. Schlie, **D. Wille**, L. Cleophas, and I. Schaefer. "Clustering Variation Points in MATLAB/Simulink Models Using Reverse Signal Propagation Analysis". In: *Proc. of the Intl. Conference on Software Reuse (ICSR)*. Vol. 10221. Lecture Notes in Computer Science. Springer, 2017, pp. 77–94.

Further peer-reviewed publications indirectly related to this thesis.

[13] K. Wehling, **D. Wille**, C. Seidl, and I. Schaefer. "Reducing Variability of Technically Related Software Systems in Large-Scale IT Landscapes". In: *Proc. of the Intl. Conference on Computer Science and Software Engineering (CASCON)*. **IBM Center for Advanced Studies Best Paper Award**. IBM Corporation, 2018, pp. 224–235.

[14] K. Wehling, **D. Wille**, C. Seidl, and I. Schaefer. "Automated Recommendations for Reducing Unnecessary Variability of Technology Architectures". In: *Proc. of the Intl. Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2017, pp. 1–10.

[15] K. Wehling, **D. Wille**, C. Seidl, and I. Schaefer. "Decision Support for Reducing Unnecessary IT Complexity of Application Architectures". In: *Proc. of the Intl. Workshop on decision Making in Software ARCHitecture (MARCH)*. IEEE, 2017, pp. 161–168.

[16] K. Wehling, **D. Wille**, M. Pluchator, and I. Schaefer. "Towards Reducing the Complexity of Enterprise Architectures by Identifying Standard Variants Using Variability Mining". In: *Automobil Symposium Wildau*. Technische Hochschule Wildau, Germany, 2016, pp. 37–43.

Contents

List of Abbreviations

AOP . . . . . .Aspect-Oriented Programming

API . . . . . .Application Programming Interface

ASP . . . . . .Active Server Page

AST . . . . . .Abstract Syntax Tree

ATL . . . . . .Atlas Transformation Language

BFS . . . . . .Breadth-First Search

CMOF . . . . . .Complete Meta-Object Facility

CSV . . . . . .Comma-Separated Values

CVL . . . . . .Common Variability Language

DFA . . . . . .Deterministic Finite Automaton

DFG . . . . . .Deutsche Forschungsgemeinschaft

DFS . . . . . .Depth-First Search

DOP . . . . . .Delta-Oriented Programming

DSL . . . . . .Domain Specific Language

EMF . . . . . .Eclipse Modeling Framework

EML . . . . . .Epsilon Merging Language

EMOF . . . . . .Essential Meta-Object Facility

EOL . . . . . .Epsilon Object Language

ETL . . . . . .Epsilon Transformation Language

FBD . . . . . .Function Block Diagram

FOP . . . . . .Feature-Oriented Programming

FST . . . . . .Feature Structure Tree

GPL . . . . . .General Purpose Language

GUI . . . . . .Graphical User Interface

HAC . . . . . .Hierachical Agglomerative Clustering

HTML . . . . . .Hypertext Markup Language
IDE . . . . . .Integrated Development Environment
IDF . . . . . .Inverse Document Frequency
IR . . . . . .Information Retrieval
LOC . . . . . .Lines of Code
MDSD . . . . . .Model-Driven Software Development
MIL . . . . . .Model-in-the-Loop
MOF . . . . . .Meta-Object Facility
MSPL . . . . . .Multi Software Product Line
NLP . . . . . .Natural Language Processing
OCL . . . . . .Object Constraint Language
OMG . . . . . .Object Management Group
OVM . . . . . .Orthogonal Variability Model
PDG . . . . . .Program Dependence Graph
QVT . . . . . .Query/View/Transformation
RCP . . . . . .Rich Client Platform
SPL . . . . . .Software Product Line
SPLE . . . . . .Software Product Line Engineering
TXL . . . . . .Turing Extender Language
UML . . . . . .Unified Modeling Language
URI . . . . . .Uniform Resource Identifier
UUID . . . . . .Universally Unique Identifier
VCS . . . . . .Version Control System
VSM . . . . . .Vector Space Model
VSpec . . . . . .Variability Specification
XMI . . . . . .XML Metadata Interchange
XML . . . . . .Extensible Markup Language
XVCL . . . . . .XML-Based Variant Configuration Language

Part I.

Context and Preliminaries

1 Introduction

Most innovations in modern systems are enabled by software. As a result, the amount of developed software functionality and its importance for companies drastically increases. For example, more than 80% of innovations in modern cars are developed through software and, thus, the corresponding systems are running up to one gigabyte of software [BKP+07]. As a result, industry faces an increasing number of challenges regarding functionality, efficiency, reuse, and reliability of developed software [Bro06, Kni02, BKP+07]. Especially in safety-critical domains, such as the automotive domain or the aviation domain, these challenges are linked with large additional costs [Bro06, Kni02, BKP+07]. One of the driving factors for these costs are legal regulations, which force companies to certify their products and to follow development approaches based on standards, such as the IEC 61508 [Int10] and its adoptions for the automotive domain as the ISO 26262 [Int11] or for the aviation domain as the DO-178C [Rad12]. Furthermore, the high competition and demand for shorter development cycles increases the pressure on companies to develop their software systems in an efficient way [BKP+07].

1.1. Motivation

Companies attempt to overcome the challenges in software development by abstracting from the concrete problems using different approaches to execute development of functionality on a more manageable level [BOJ04, DHJ+08]. One common strategy is application of *generative programming* to develop systems using *model-based languages*, such as The Mathworks Matlab/Simulink[1] or different statechart notations, and automatically generate large parts of the code [BOJ04, DHJ+08, WM95, CE00, SV06]. For instance, most of these languages allow to use hierarchical decomposition to start development of systems on a high level of abstraction and refine their functionality with each added hierarchy level [BOJ04, DHJ+08]. Because of this high level of abstraction and their general design, these languages allow to develop systems on a more understandable and less complex level [BOJ04, WM95]. Overall, generating code through well-tested and, ideally, certified code generators has the advantage of reducing the errors that are potentially introduced by manually editing low-level source code [BOJ04].

However, in addition to the high complexity inherently present in safety-critical domains, companies often are faced with development of families of software systems [PBK+07, BOJ04]. Driving factors are the customers' specific requirements regarding the developed systems [PBK+07]. For example, one customer of a car is interested in a high-end driver assistance system, while another one is not willing to pay for such additional functionality. In addition, legal requirements in different countries have an impact on the development of products and require companies to adapt the systems' functionality (e.g., varying emergency call systems are used in different countries) [PBK+07]. Thus, while model-based languages and generative programming help to alleviate the problems

[1] https://www.mathworks.com/products/simulink.html

linked with the complexity inherent in different target domains, developing corresponding variants of systems (i.e., software with largely similar functionality that is only varying in small parts) is still a time consuming and costly task. In most cases, the functionality of developed systems is too large and too complex to allow a complete redevelopment of their functionality for each variant.

As a result, copying existing software solutions and modifying them to changed requirements is common practice, especially for domains with large numbers of product variants (e.g., the automotive domain) [DRB+13, PBK+07]. These so-called *clone-and-own* approaches are an efficient means to create new variants from existing products as the new software does not have to be developed from scratch, but reuses a large set of existing functionality [LT03, DRB+13].

However, while these clone-and-own approaches help companies to reduce the initial overhead when creating new software variants, they involve risks in the long run and are considered to be harmful for the long-term development process [DRB+13, KG06]. Over time, the number of related variants and the associated maintenance effort grows as in most cases the relations between created variants is not documented and only is implicitly known by domain experts [DRB+13]. This is especially problematic if knowledgeable developers leave the company (e.g., due their retirement) and relevant information for efficient maintenance of variants is lost. As a result, fixing an identified error in all variants of the software family becomes a complex task because each variant has to be manually analyzed in order to identify the affected parts [DRB+13]. Furthermore, updating software parts to a new version might involve additional reimplementation effort when substantial changes were executed during the initial clone-and-own phase [DRB+13].

While a variety of approaches exist to support developers in work with clone-and-own strategies [RC13a, RCC13, RCC15, FLL+15, LLE17, LFL+15, JBA+15, DKZ+12, PTS+16], these approaches often can only ease the associated problems and do not solve them completely. For instance, Rubin et al. provide a conceptual framework with different operators to support developers during maintenance of cloned variants. By implementing the operators for specific scenarios, companies can, for example, help developers to identify source code for a product feature and corresponding dependencies to other product features [RC13a, RCC13, RCC15]. However, while understanding code for cloned variants and maintaining them becomes easier, code duplication across variants and possible divergence of code still remains a problem. Overall, maintaining a large set of related software variants, which evolved using clone-and-own techniques remains a costly and tedious task.

To overcome such problems and allow for structured reuse of functionality across variants, academia promotes *Software Product Lines (SPLs)* [PBL05, CN01, CE00]. SPLs consist of the common core functionality for all variants and allow definition of additional functionality in form of reusable artifacts. Using corresponding generation facilities, it is possible to configure different variants and automatically generate them from these artifacts [PBL05, CN01, CE00]. Major advantage of this approach is that all implementation artifacts are managed in a single repository. As a result, errors can be fixed by editing these artifacts and regenerating all affected variants. Overall, the quality of developed code increases and the time to market for new products is reduced [PBL05, CN01].

Adopting SPLs as a reuse strategy is successfully demonstrated by a variety of companies [LSR07, PBL05, JDB07, JB09, SH04, BCK03] and the *SPLC Hall of Fame* at the *International Systems and Software Product Line Conference (SPLC)*[2] promotes such successes to provide guidance and motivation to interested companies. However, this step is often not planned when starting development of soft-

[2] http://www.splc.net/hall-of-fame/

ware, but only after companies realize that variants of their software are needed and they already created different variants in an ad-hoc manner by using clone-and-own approaches. In such cases, an *extractive adoption strategy* is often used to achieve reuse of functionality from existing variants in an SPL and to manage their variability [BRN+13]. However, creating an SPL from existing variants involves large effort as developers have to identify reuse potential for these variants to enable efficient design of the created SPL. This is especially problematic in industry were often large models are used to develop complex systems [Bro06, BOJ04]. For example, Beine et al. [BOJ04] speak of Matlab/Simulink models with more than 100,000 hierarchical elements comprising further functionality in their hierarchy. In addition, the identified variability has to be manually encoded in reusable artifacts to enable correct generation of all needed variants. Altogether, manual analysis of potentially huge sets of model variants comprising such large numbers of artifacts and encoding the identified variability in an SPL is infeasible [LBL+15].

1.2. State of the Art and Problem Statement

For a migration of existing models to an SPL, developers have to identify the common and varying parts amongst the analyzed variants. In the literature, a variety of approaches exists to support identification of such information. On the one hand, *clone detection* algorithms, such as [ACD+12a, ACD+12b, ASH11, DHJ+08, PNN+09], exist to identify cloned (i.e., common) parts of model variants. On the other hand, *differencing* algorithms, such as [KKT11, KWN05, LGJ07, MRR11a, MRR11b, XS05], exist to identify differences between model variants by identifying their common parts first and using them to derive the differences afterwards. While both clone detection and differencing algorithms are in general capable of identifying relevant variability information for the migration to an SPL, they normally do not allow extraction of this variability to reusable artifacts. As a result, they are not directly applicable for migrating existing model variants to an SPL.

To overcome these limitations, different authors proposed a variety of *reverse-engineering algorithms* to automatically identify and migrate variability between related model variants to SPLs.

Generic Reverse-Engineering Algorithms exist to identify variability relations between model variants relying on generic representations of the models [FBH+15, MZK+14, MZB+15a, ZHM11]. For instance, the approach by Zhang et al. [ZHM11] completely relies on an existing generic comparison algorithm for models that were developed using the Ecore modeling notation within the *Eclipse Modeling Framework (EMF)*. Based on the identified variability relations, all approaches are capable of creating corresponding SPLs enabling the derivation of variants. By comparing the models on a generic level, the approaches are not dependent on language-specific or domain-specific details. As a result, generic reverse-engineering algorithms are capable of identifying common and varying parts for a large variety of modeling notations and enable general SPL migration support for them.

Language-Specific Reverse-Engineering Algorithms exist that allow language-specific identification of variability relations [ARS+14, NSC+07, NSC+12, RC12, RC13c, RC13d, RPK10, RPK12a]. For instance, the approach by Nejati et al. [NSC+07, NSC+12] was specifically designed to identify variability relations for statecharts by applying corresponding metrics to calculate the similarity between their states. Based on identified matches between compared variants, the approach merges a single model with presence annotations for the model elements in different variants. As a result, these language-specific approaches focus directly on enabling SPL migration support for a single language.

As we have seen a variety of approaches exist to support extractive migration of existing model variants to corresponding SPL realizations. While these approaches are in general capable of identifying *variability relations* (i.e., common and varying parts) between analyzed variants and encoding them in SPL realizations, we identified the following problems:

- *Too Generic or Too Specific:* On the one hand, some of the existing approaches rely on *generic* identification algorithms to compare model variants on a high level of abstraction [FBH+15, MZK+14, MZB+15a, ZHM11]. While this enables application of the algorithms to a variety of modeling languages, it gives developers only limited ways to influence the executed comparisons (e.g., to execute language-specific analysis) and adapt them to problem-specific scenarios (e.g., the requirements of a specific domain). As a result, the identified variability relations might not always conform with the expectations of the executing domain experts, because important details were not appropriately considered during the comparisons.

  On the other hand, some of the existing approaches represent solutions that are *specific* to a certain language or domain [ARS+14, NSC+07, NSC+12, RC12, RC13c, RC13d, RPK10, RPK12a]. While the general ideas of the approaches might in theory be applicable to other languages or domains, no corresponding tooling exists to support developers during an adaptation for their problems. As a result, these approaches are solely applicable for their intended purpose.

  Due to the aforementioned problems, the existing approaches are either too generic to allow application for domain-specific problems [FBH+15, MZK+14, MZB+15a, ZHM11] or too specific for a certain language or domain to allow general applicability for other languages or domains [ARS+14, NSC+07, NSC+12, RC12, RC13c, RC13d, RPK10, RPK12a].

- *No Explicit Low-Level Variability Relations:* While all approaches are generally capable of identifying variability relations in their corresponding application areas, they only highlight in which variants each model element is contained. However, we believe that additional information on the explicit variability relations between model elements is inevitable to fully understand how the variability between variants is developed and to enable corresponding discussions amongst developers. Such explicit variability relations highlight which model elements (e.g., states or transitions in statecharts) are *mandatory* (i.e., contained in all variants), *optional* (i.e., only contained in some of the analyzed variants) or *alternative* (i.e., mutually-exclusive to other model elements). As a result, with such details missing from the variability information identified by existing approaches, developers cannot directly see these relations and have to spent additional effort on identifying them.

- *High-Level Relations have to be Known Upfront:* All of the existing approaches implicitly assume that companies are aware of the high-level relations between existing model variants. For example, larger clusters of highly related variants might exist that are less similar to variants in other clusters. In addition, outlier variants might exist that have only a very low or even no connection at all to the remaining variants (e.g., in cases where the variants diverged too much due to completely different requirements). However, due to the lacking documentation in clone-and-own scenarios, we highly doubt that information on such relations is available in all companies. As a result, all of the existing approaches are not directly applicable in such scenarios as these relations have to be identified during tedious manual analysis of all variants.

In summary, we see the missing support for configuring existing algorithms to domain-specific problems or adapting them for new languages as a major issue. As a result, applicability is limited to specific cases where these algorithms yield the desired results and adopting managed reuse of variability in an SPL remains a challenge. Furthermore, we believe that the assumption about companies being aware of all relations between their existing variants is dangerous and in most cases wrong. Thus, huge manual effort might be needed to identify relevant variants for a migration to an SPL in large industrial model repositories.

1.3. Research Questions

From the above problem statement, we derive the following central research question of this thesis:

> *How can we semi-automatically migrate a set of existing model variants with possibly unclear relations to a software product line and make identified variability relations explicit to domain experts?*

We identified the following three research questions, each targeting one of the related key aspects:

Research Question RQ1 – Identifying High-Level Variability Relations *How can we achieve a generic approach to identify high-level relations between large sets of models, whose relationships are unclear?*
This research question focuses on finding an approach to analyze potentially large sets of model variants, whose relations are not clear because of a variety of possible reasons (e.g., unclear documentation or retirement of senior developers). Focus is not a precise and in-depth analysis, but an approximate comparison of variants to get a general overview of their high-level relations.

Research Question RQ2 – Identifying Low-Level Variability Relations *How can we achieve an adaptable and configurable approach to identify detailed low-level relations between related model variants?*
This research question focuses on finding an approach to identify detailed and explicit low-level relations between elements of related model variants. Focus is adaptability and configurability of the found solutions to allow for an easy adaptation for a) different situations (e.g., changed company goals) and b) different modeling languages.

Research Question RQ3 – Migrating Existing Variants to an SPL *How can we achieve an easy and semi-automatic migration of existing model variants to a corresponding software product line realization?*
This research question focuses on finding an approach to migrate a set of existing model variants based on identified low-level variability relations to a corresponding software product line. Focus is to dissect the existing model variants into cohesive parts constituting their variability and storing them in reusable software product line artifacts allowing generation of all input variants.

1.4. Overall Approach

The approach developed in this thesis consists of three phases, which allow developers a controlled and well-informed transition from cloned product variants to an SPL comprising the necessary realization artifacts for their common and varying parts. Main focus of our techniques lies on the adaptability for different modeling languages and scenarios while providing reliable and scalable results at the same time. To this end, we present generic algorithms that allow easy adaptation for developers and lower the barrier for practitioners to migrate existing variants to an SPL.

In Figure 1.1, we show the general workflow of our approach. The complete approach relies on model-based techniques to achieve the described adaptability and generality of the algorithms. Thus, prior to processing their product variants, the developers have to define a language-specific meta-model for the analyzed language and import their product variants into this representation.

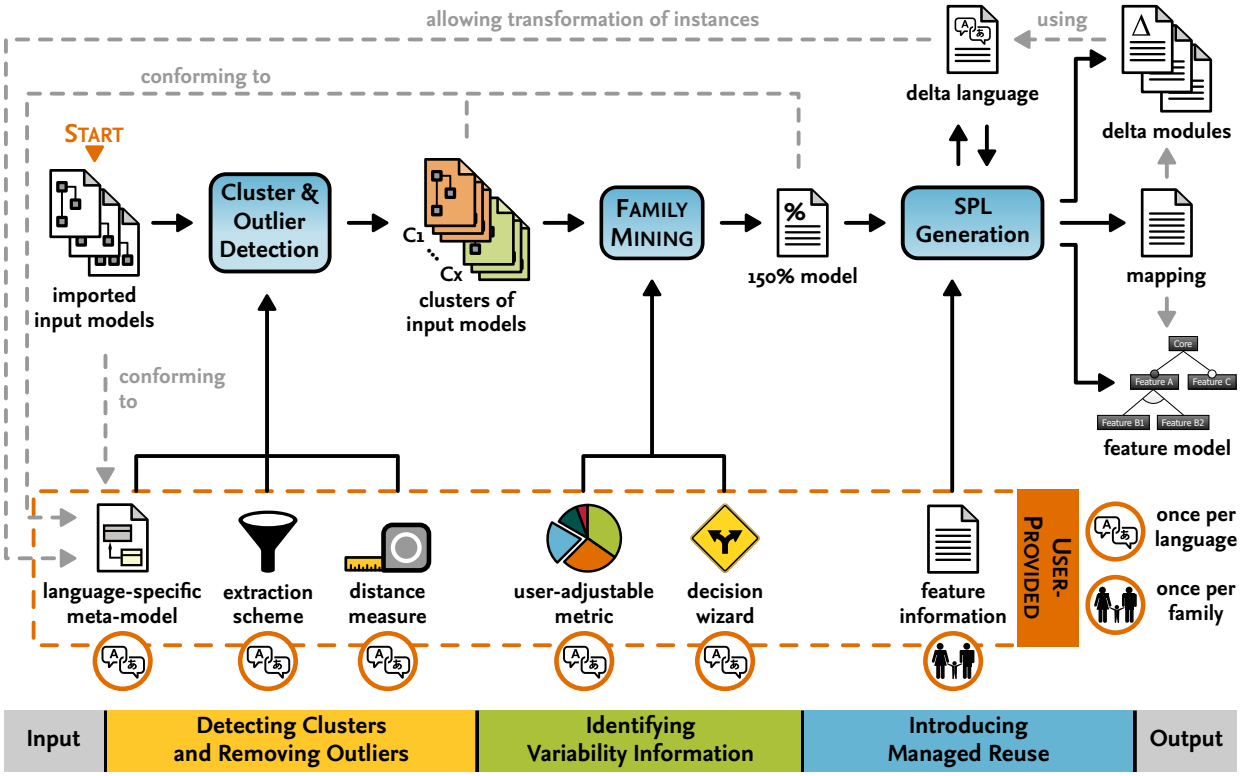


Figure 1.1.: Overall workflow of the approach for a *Custom-Tailored Product Line Extraction*.

Phase 1 – Detecting Clusters and Removing Outliers The first phase processes the imported product variants and allows identification of outliers and clusters for the input products. While outliers represent products with a very low similarity compared to all other variants, clusters might indicate sets of products that have a high similarity according to their implementation (e.g., they realize similar functionality). To this end, the corresponding techniques rely on language-specific extraction schemes and distance measures to identify and compare relevant information for a fast and approximate high-level analysis of the models. Overall, this phase is particular helpful in situations where developers have to get a quick overview of a large number of variants whose relations are unknown (e.g., after acquiring a company and integrating their models in the common repository).

Phase 2 – Identifying Variability Information The second phase executes the so-called Family Mining to identify detailed low-level variability information in form of explicit relations between the compared model elements and their concrete variability (i.e., whether they are common or varying across the variants). The approach relies on language-specific metrics that allow easy adjustment of the algorithms for user-specific scenarios and decision wizards to support the identification of relations for similar model elements. The results are stored in a 150% model comprising the model elements from all analyzed variants together with detailed variability information annotated to them.

Phase 3 – Introducing Managed Reuse The third phase processes the identified variability information in the created 150% model to generate corresponding SPL artifacts. To this end, the approach allows to derive a delta language providing delta operations that are specifically tailored towards the used meta-model and allow transformation of existing variants by adding, removing or modifying model elements. Using this delta language (or a user-provided one), the approach generates delta modules encoding the identified variability in corresponding delta modules. By providing additional details on the features (i.e., cohesive functionality) in the analyzed product family, the user can support this approach and allow dissection of the existing functionality into reusable parts in corresponding delta modules that can later be selected during configuration of variants.

1.5. Structure of the Thesis

This thesis is divided in three parts and structured as follows:

Part I – Context and Preliminaries This part comprises the introduction with a motivation of the topic and a discussion of the state of the art for this thesis. Based on a clear problem statement, we define our research questions and show an overview of the approach developed within this thesis. Chapter 2 introduces a running example used to discuss the developed approach. Furthermore, we discuss development of software systems comprising variability and the fundamental techniques used as an underlying basis for our approach.

Part II – Custom-Tailored Product Line Extraction This part consists of three chapters describing the approach developed in this thesis to identify variability relations between large sets of model variants and migrating them to an SPL. Chapter 3 describes an algorithm, which is capable of identifying clusters of highly related model variants and to identify outlier variants, which have only low or even no similarity to other variants. Using this algorithm it is possible to get an initial overview of the high-level relations between existing model variants. Chapter 4 describes an algorithm to identify detailed low-level variability relations between model variants. The algorithm is highly adjustable to new settings and can easily be adapted for different languages using our tooling. Chapter 5 describes an algorithm to migrate model variants based on their low-level variability to a delta-oriented SPL. By providing additional details on the existing variants, users can support the variants' dissection into separate delta modules with mappings to the configuration options of the created SPL.

Part III – Realization and Application This part consists of three chapters describing the realization and application of the developed algorithms. Chapter 6 describes the implementation of all approaches developed in this thesis. Chapter 7 describes a detailed evaluation of the developed approaches using two publicly available case studies with industrial background and one case study provided by our industry partner. In addition, we show results from expert interviews, which we executed with engineers from our industry partner to collect feedback on the applicability of our approaches in industry. Chapter 8 concludes this thesis with a discussion of the results and outlook to possible future application areas.

2 Background

In this chapter, we discuss the background of this thesis in detail. First, we explain model-driven software development as an underlying technique for the algorithms developed in this thesis (cf. Section 2.1). Next, we introduce two running examples consisting of two statechart variants each (cf. Section 2.2) to explain the ideas and algorithms developed in the course of this thesis. Finally, we describe the general details of variability management in software system variants in the context of software product lines and discuss possible adoption strategies (cf. Section 2.3).

2.1. Model-Driven Software Development

Model-Driven Software Development (MDSD) applies *models* not only for documentation purposes (e.g., class diagrams), but considers them as first-class development artifacts similar to source code, because they allow direct source code generation [SV06]. In this context, we use the following definition for models.

Definition 2.1: Models

"A *model* is an abstract representation of a system's structure, function or behavior." [SV06]

This definition describes two of the key aspects provided by MDSD:

- *Abstraction:* Using different strategies modern model-based development languages allow to *abstract* from complex problems and provide developers with means to approach them step-by-step. For instance, many languages allow to use *hierarchical refinement* to decompose solutions into smaller, more manageable parts. Starting at a high level of abstraction (e.g., a single element with the developed function's name), it is possible to refine the implementation with each added hierarchy level (e.g., adding further sub components and implementing them).
- *Simplified and Focused Representation:* Furthermore, models allow simplified development of complex functionality as they solely focus on the concrete goals of modeled systems and omit unnecessary details.

The generative capabilities and the high degree of abstraction are only two of the reasons why MDSD enjoys great popularity in industry [BOJ04, Bro06]. Further drivers are legal requirements, such as the IEC 61508 [Int10] with its adoptions for the automotive domain as the ISO 26262 [Int11] or for the aviation domain as the DO-178C [Rad12]. Such development norms force companies in safety-critical domains to certify their products and prove reliability of developed systems and code. To reduce the corresponding challenges and the linked effort, it is common practice to apply code generation using already well-tested and, ideally, certified code generators which proved to generate reliable code [BOJ04]. As a result, the overall code quality of developed systems and their development times typically improve [SV06]. Furthermore, MDSD allows developers to focus on

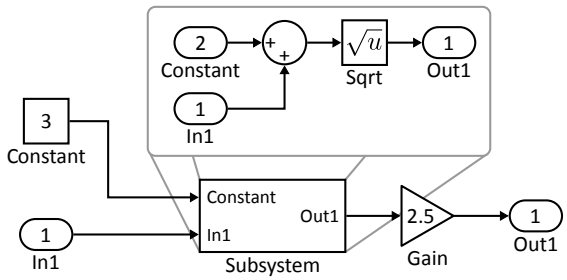


Figure 2.1.: Exemplary Matlab/Simulink model.

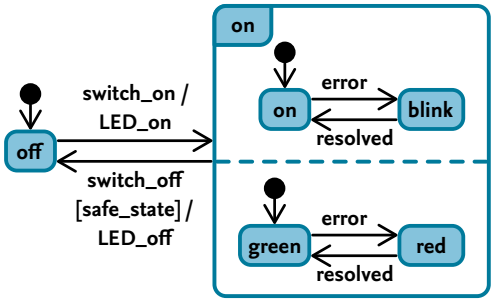


Figure 2.2.: Exemplary statechart.

implementing the business logic rather than spending time on finding errors [SV06]. *Model-in-the-Loop (MIL)* approaches allow companies to test developed functionality by executing the models in a simulated productive environment (e.g., the environment of a car) [Plu06]. Thus, time and money can be saved by executing such tests prior to the code generation.

Using *model transformations* it is possible to transform developed models into other representations and, thus, allow flexibility in the design and development of systems. For example, it is possible to use *model-to-model transformations* during evolution of the developed system to update all occurrences of a certain model element. In addition, *model-to-text transformations* allow to generate executable source code for the developed models.

Block-Based Modeling Languages Due to the advantages of MDSD, a large variety of modeling languages exist to support development in different domains. Examples are the The MathWorks Matlab/Simulink and The MathWorks Stateflow[1] for the automotive domain. In this thesis, we focus on developing algorithms to identify and exploit variability information for models that were developed with languages that we define as block-based modeling languages.

Definition 2.2: Block-Based Modeling Languages

Models developed with *block-based modeling languages* consist of structures similar to directed graphs. Their complete behavior is defined by the functionality of used blocks (i.e., the graphs' nodes) and connections (i.e., the graphs' edges) between them.

This definition applies to most industrial modeling languages and the capabilities of their language elements (i.e.,nodes and edges) differs depending on the used development paradigm. As we discuss all generic concepts developed in this thesis with respect to Matlab/Simulink models and statecharts, we give a short introduction to both languages in the following two paragraphs.

Matlab/Simulink Matlab/Simulink represents a *dataflow-oriented* modeling language. Thus, it is used to model dataflows by using *atomic functions* in form of reusable *blocks* from the Matlab/Simulink library to transform input signals to generate corresponding outputs. The signals between the blocks are emitted using *connectors* linking their *ports*. These ports define the blocks' *interfaces* (i.e., their incoming and outgoing connections). In Figure 2.1, we show an exemplary Matlab/Simulink model. In this example, a signal is introduced to the system via the *Inport Block* `In1` and together with the constant value 3 from the *Constant Block* `Constant` passed to the *Subsystem Block*

[1] `https://www.mathworks.com/products/stateflow.html`

`Subsystem`. Such *subsystems* allow hierarchical decomposition of models and can be arbitrarily nested to efficiently develop complex functionality. In this example, the input signal and the constant are summed up and the square root is calculated using the *Square Root Block* `Sqrt`. The result from the `Subsystem` is amplified with the constant value 2.5 using the *Gain Block* `Gain` and emitted via the *Outport Block* `Out1` for further processing (e.g., to trigger the actuator of a car).

Statecharts In contrast to MATLAB/SIMULINK, statecharts represent a *state-based* modeling notation and exist in different flavors. Statecharts represent a notion to model the discrete states of a system using *Deterministic Finite Automatons (DFAs)* [HMU06]. However, depending on the used notation, their syntax and semantics are often extended with additional elements [Har87]. In general, statecharts use a finite set of *states* to model the system and have a single *initial state* to deterministically identify the system start. Unlike MATLAB/SIMULINK, the behavior of statecharts is defined mostly through their *transitions* linking the system states. These transitions are triggered by *events* and change the system state in case their *guards* (i.e., logical conditions) are fulfilled. During the transition from one state to another, these transitions allow execution of *actions*, which in turn might trigger further transitions through corresponding events. In Figure 2.2, we show an exemplary statechart to model the behavior of an LED showing the status of a system. The system starts in the `off` state and changes in the `on` state if the `switch_on` event is triggered (e.g., through a button). In contrast to MATLAB/SIMULINK, different statechart notations not only allow modeling of hierarchical decomposition, but also of *parallel execution* (i.e., the system can be in multiple states at once). In Figure 2.2, the `on` state represents such a *parallel state* and the dashed line shows the separation between the *regions* modeling its behavior. While the upper region uses the lighting of the LED to show the status of the system, the lower region uses the color of the LED. Due to the parallel modeling, the LED can be either in the {`on`, `green`} state or the {`blink`, `red`} state. Normally, the LED is in the first state, but changes to the second state if an `error` in the system occurs. After the error is `resolved`, the system switches back to the first state. The LED can only be switched off (i.e., the `LED_off` action is executed) if the `switch_off` event is triggered (e.g., through a button) and the system is in a safe state (i.e., the guard checking the `safe_state` evaluates to true).

Meta-Modeling To define the concepts that can be used in a model describing a system, it is possible to define meta-models.

Definition 2.3: Meta-Models

"A meta-model defines, [...], the basic constructs that may occur in a concrete model." [SV06]

Such a meta-model can be seen as a language definition for the used modeling language, similar to a grammar for a textual programming language. As a result, each developed model is an instance of a language-specific meta-model and uses the contained concepts. These general concepts are captured by the *Object Management Group (OMG)* in the *Meta-Object Facility (MOF)* [Obj16a] to standardize the development of software using meta-modeling. The MOF defines four layers *M0* to *M3*. In Figure 2.3, we use a statechart implementation of a finger protection system to explain them. Layer *M0* describes *real world objects*, i.e., in this example the physical finger protection system. Layer *M1* constitutes a model of these real world objects, i.e., in this example a statechart implementation describing the functionality of the finger protection system. Layer *M2* defines the meta-model with all language-specific concepts and, thus, is used by the layer *M1*. Finally, the layer

M3 defines the meta-meta-model of the MOF that serves as basis for defined meta-models on layer *M2* and comprises the general concepts used for meta-modeling. All concepts of the *M3* are defined using concepts from layer *M3* itself to reduce the number of introduced meta-levels.

The OMG defined all necessary model entities and their possible relations for layer *M3* in the *Complete Meta-Object Facility (CMOF)*. Based on this comprehensive standard, the OMG also defined the *Essential Meta-Object Facility (EMOF)* to reduce the number of used concepts to the most essential model entities and relations. These elements can be seen in Figure 2.4.

A notation close to the EMOF standard is the Ecore meta-meta-model of the *Eclipse Modeling Framework (EMF)*[2], which offers a large variety of tools and extensions to efficiently allow model-based development in Eclipse[3]. For instance, it is possible to apply additional techniques to develop textual languages based on meta-models by using the Xtext framework[4] or EMFText[5]. The Ecore notation is also the underlying basis of our implementation in this thesis.

Using developed models or meta-models it is possible to execute generative programming techniques [CE00] and, for example, derive Java Source Code from meta-models that were created with the EMF. Furthermore, it is easily possible to define formal constraints and to verify whether developed models conform to their parent meta-model. For example, the *Object Constraint Language (OCL)* is part of the *Unified Modeling Language (UML)* standard developed by the OMG [Obj15] and allows definition of such constraints for corresponding models.

The solutions in this thesis rely on MDSD to achieve an abstraction from concrete modeling languages and, thus, to support our solutions to a variety of different languages. By using a common base meta-model as a basis for our algorithms, we are able to exploit MDSD principles and to allow exchangeability of the concrete language-specific meta-models.

2.2. The Body Comfort System as Running Example

To demonstrate and discuss the solutions described in the course of this thesis, we introduce two running examples in the context of the automotive domain. For this purpose, we use statecharts implementing the functionality for the *Body Comfort System (BCS)* of a car. The presented models originate from the *BCS* case study [LLL+13], which was developed based on a real-world car that never went into production and comprises 27 product features with constraints between them. Exemplary features are a Power Window (PW) in different implementations, a corresponding Finger Protection (FP) during movement of the window, a Central Locking System (CLS) and Exterior Mirrors (EM) that optionally can be equipped with Heating to deice them in winter.

Please note, that the *BCS* case study provides us with model variants to evaluate the results of this thesis. However, all ideas and algorithms presented in this thesis were developed independently from the concrete implementation of the *BCS* and, thus, are not biased by an overfitting for the concrete scenarios of the *BCS*. As a result, the presented models serve as a running example used for illustrative purposes.

While the size of industrial models can be enormous [Bro06, BOJ04], the following running examples only constitute a small number of possible model elements (i.e., states and transitions).

[2] https://www.eclipse.org/modeling/emf/
[3] https://www.eclipse.org/
[4] https://www.eclipse.org/Xtext/
[5] http://www.emftext.org

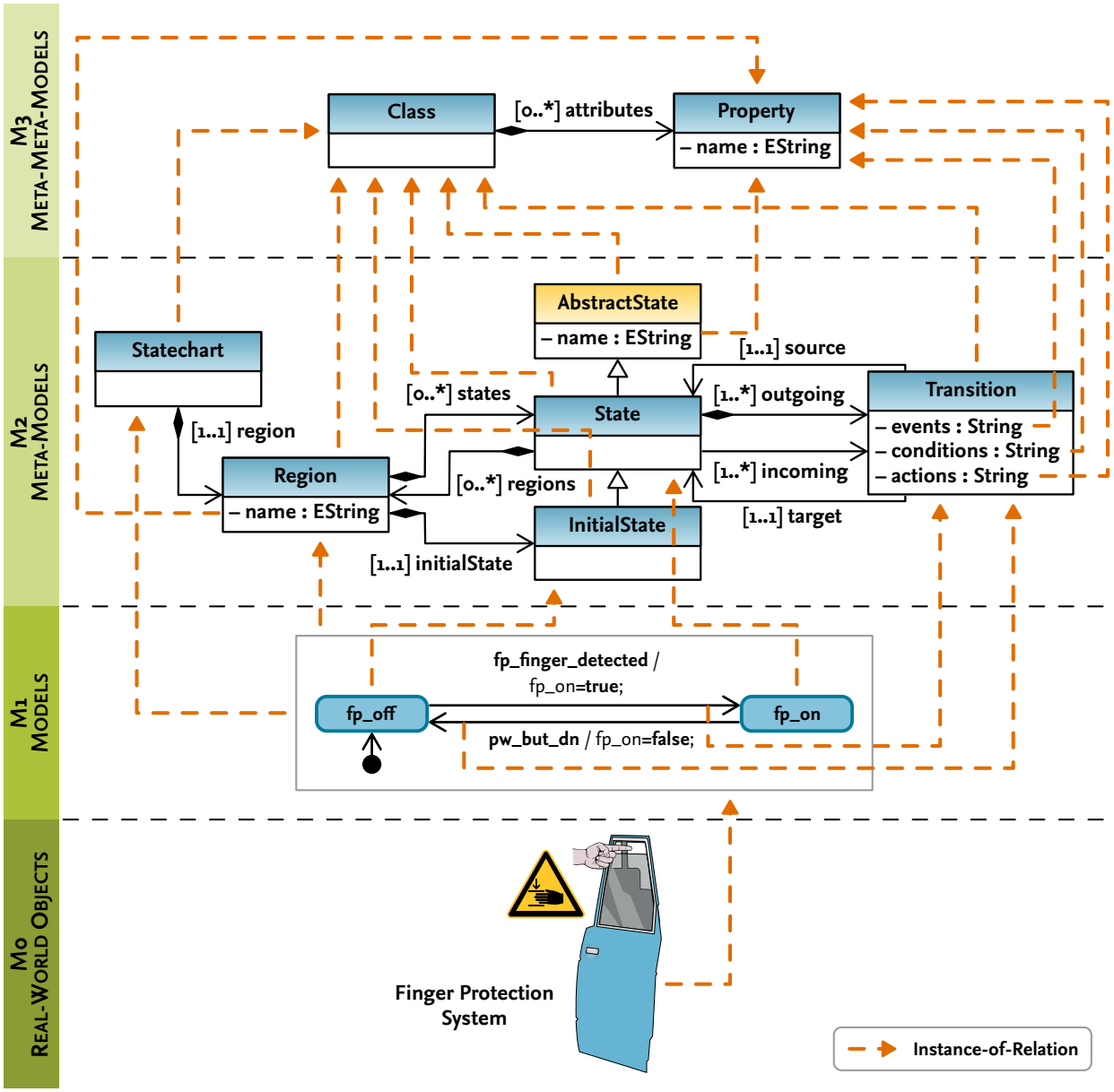


Figure 2.3.: The layers of the MOF illustrated by the statechart implementation of a finger protection system.

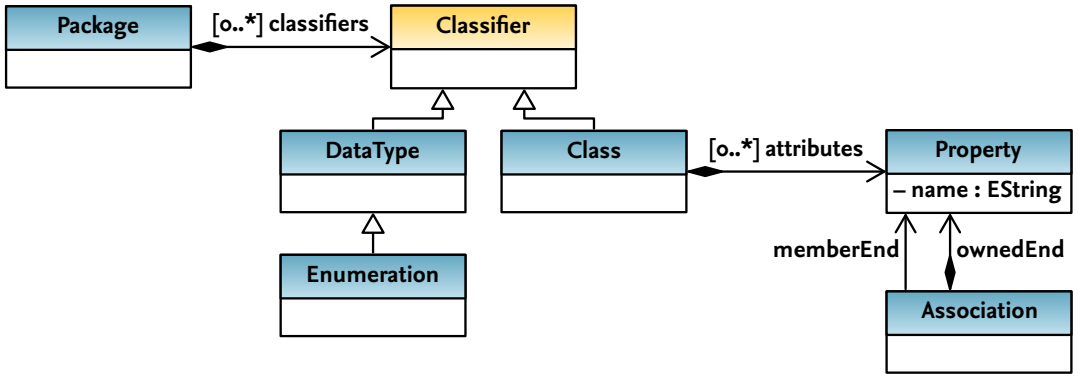
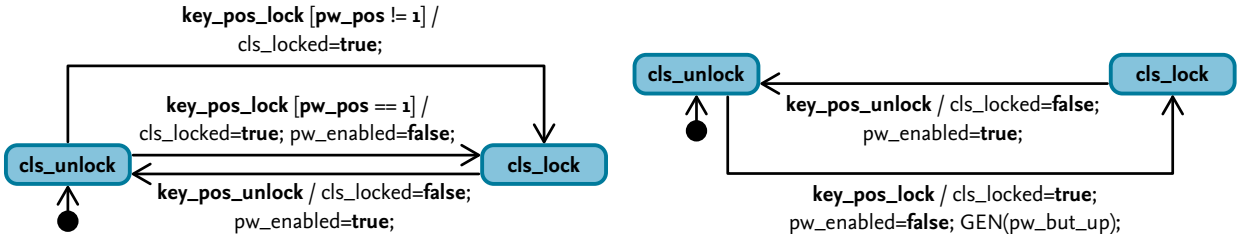


Figure 2.4.: The entities of the EMOF reducing the CMOF meta-meta-model to its essential entities.



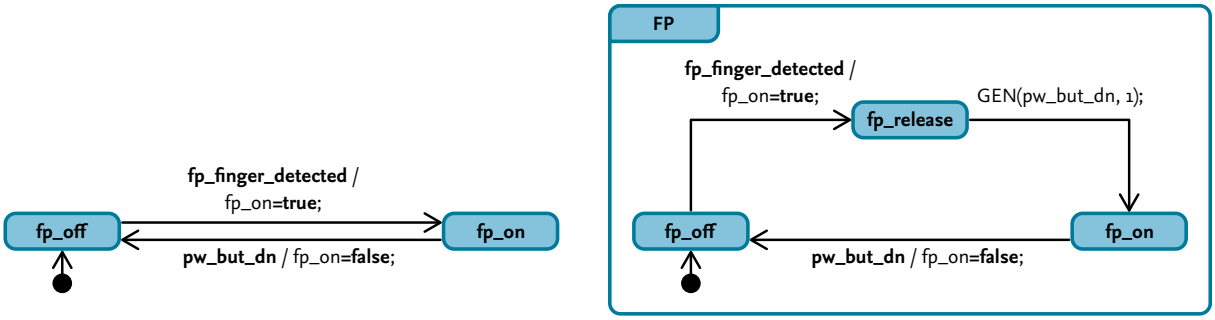
(a) With a Manual Power Window (ManPW). (b) With an Automatic Power Window (AutoPW).

Figure 2.5.: Two variants of a Central Locking System (CLS) of a car with differing Power Windows (PWs).

However, they were specifically selected to allow demonstration of the solutions developed in this thesis using easily graspable (i.e., due their limited size) and realistic (i.e., due to the real-world background) models. For a detailed evaluation of the developed solutions, we additionally show their applicability to industrial-scale models in the evaluation of this thesis (cf. Chapter 7).

Running Example 1 The first running example consists of two variants of the Central Locking System (CLS) feature that depends on the used Power Window (PW), which can either be realized as a Manual Power Window (ManPW) or an Automatic Power Window (AutoPW). In Figure 2.5, we present the corresponding statechart implementations, both realizing their functionality using two states `cls_unlock` and `cls_lock` showing whether the car is currently locked. The major difference between the two variants manifests itself when the user locks the car (i.e., triggered through the `key_pos_lock` event). The CLS depending on the ManPW (cf. Figure 2.5a) distinguishes for this transition between a case where the window was previously closed (i.e. `[pw_pos == 1]`) and a case where the window is still open (i.e., `[pw_pos != 1]`). For the former case, the CLS automatically disables the PW to prevent theft (i.e., through the `pw_enabled=false` action). In the latter case, the PW is still enabled to allow drivers to still close it (e.g., when sitting inside the locked car). In contrast, the CLS depending on the AutoPW (cf. Figure 2.5b) disables the PW (i.e., through the `pw_enabled=false` action) in any case, but generates a event to automatically close any opened windows (i.e., `GEN(pw_but_up)`). In the course of the thesis, this running example allows us to demonstrate the developed solutions for automatic identification of variability relations (i.e., common and differing parts) between developed models.

Running Example 2 The second running example consists of two variants of the Finger Protection (FP) feature. In this case, a developer decided to copy the initial version of the FP in Figure 2.6a to develop the variant in Figure 2.6b by encapsulating its functionality in a hierarchical state called `FP`. This allows higher encapsulation and might ease reuse for further variants. Both variants use the states `fp_off` and `fp_on` to show whether the FP is active or not. In case a finger is detected, the FP is active until the PW down button is pushed (i.e., to generate the `pw_but_dn` event). During the development of the second variant, the developer changed the behavior of the statechart by inserting the `fp_release` state and replacing direct connections from state `fp_off` to `fp_on` with corresponding transitions routing over the intermediate state. This way the FP is enabled in the `fp_release` state and the statechart directly triggers the generation of the `GEN(pw_but_dn, 1)` command to open the window one centimeter to release the finger. In the course of the thesis, this additional running example allows us to discuss the challenge of automatically detecting moved model elements (e.g., between hierarchy levels) and to demonstrate our corresponding solutions.



(a) Simple variant of the Finger Protection (FP).

(b) Finger Protection (FP) with automatic release and encapsulated in a hierarchical state.

Figure 2.6.: Two variants of a Finger Protection (FP) for a Power Window (PW) of a car.

2.3. Variability in Families of Software Systems

Over the last two centuries, the way products are produced changed due to a number of factors. Most importantly, with the industrial revolution and subsequent changes in society the number of people who could afford buying different kinds of products grew. With these increasing numbers of customers, the companies were challenged to increase their production rates to satisfy the customers' demands for products. To cope with these challenges, Henry Ford introduced one of the most famous changes to the production of goods by using assembly lines in his factories. This way, Ford not only was able to produce his cars for a *mass market*, but also to decrease his production times at the same time. The modern *mass production* as we now it today was born [PBL05, CE00].

While this solution solved the problem of larger customer numbers, also the need for customization of products grew. As a result, people were not interested in generic solutions anymore, but demanded products that specifically served their needs. For example, while farmers were interested in transporting tools and goods on an open cargo area, families were interested in enough seats to accommodate their children. Thus, companies were not only faced with *mass production* of products, but also with their *mass customization* [PBL05, CE00]. To solve this additional challenge, many companies started to introduce *common platforms* sharing the essential functionality for different products [PBL05, CE00]. For instance, many modern cars are build upon a common basis that provides general functions, which are extended with product-specific features. In this context, one speaks of a *product line* or *product family* as a variety of products sharing similar functionality is build upon the common platform [PBL05]. These approaches are applied in a variety of domains, such as the aviation domain or the computer hardware domain [CE00].

Similar developments can be seen for modern software systems as related challenges are faced. Previously, the complexity of software was limited and customers either could buy an existing solution with all its functionality *or* had to order a custom solution for their problems [PBL05]. In contrast, many modern systems are developed as *software-intensive systems*, because software is key driver for innovation and allows to realize functionality that otherwise would be impossible [PBL05, CN01]. In addition, companies targeting specific markets (e.g., the automotive domain) not only are faced with the *mass production* of software, but also with the need for *mass customization* as customers request new functionality (e.g., additional driver assistance systems).

Due to these challenges regarding the customization of software systems for large numbers of customers with different requirements, companies have to develop and maintain an increasing number of products. These related products are also referred to as *software families*. For this thesis, we use the corresponding definition by Parnas [Par76], who does not distinguish between "program families" and "software families". Thus, for this thesis, we refer to them as software families.

Definition 2.4: Software Families

"Program families are defined (analogously to hardware families) as sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members." [Par76]

Individual members (i.e., products) of such software families are referred to as product variants.

Definition 2.5: Product Variants

A *product variant* (or for short *variant*) represents a valid realization of one individual member of a software family that can be distinguished from all other family members due to its specific functionality or a specific configuration.

2.3.1. Clone-and-Own

With the growing size of developed systems and the limited memory space available in embedded systems, companies have to find efficient solutions to cope with development of such software families. On the one hand, implementing a single solution per customer becomes infeasible as millions of code lines realize the functionality in highly distributed systems [CN01]. On the other hand, implementing a single monolithic solution comprising all possible functionality of the system and configuring it for each customer is not possible due to the limited memory space of embedded controllers [CN01].

Thus, to allow *mass production* of their products, companies often reuse existing implementations by *copying* them to new developed systems [FV03, EEM10, Bos00, DRB+13]. In addition, to cope with specific requirements of different customers and, thus, allow *mass customization* these copies are *modified* (e.g., to implement new functionality) [FV03, EEM10, Bos00, DRB+13]. This development practice is often referred to as clone-and-own.

Definition 2.6: Clone-and-Own

"Using *clone-and-own* a new variant is created by copying and customizing assets from an existing variant." [AJB+14]

However, literature agrees that while development of product families using the clone-and-own approach has its advantages in the short-run (e.g., fast and easy introduction of new products), it often introduces large risks and problems in the long-run if not applied carefully [DRB+13, KG06]. Mentioned key challenges are the increased effort of propagating changes between clones due to missing information on their relations and the underestimated effort of adapting cloned artifacts to new requirements [DRB+13].

2.3.2. Software Product Lines

To resolve negative effects introduced by unstructured reuse approaches, such as clone-and-own, literature advertises to apply strategies for *managed reuse* instead [PBL05, CN01, CE00]. General idea is to plan reuse a priori and to make use of *managed variability* by using a common platform for developed variants and extending them with product specific functionality. In the context of software systems, variability can be distinguished in *variability in space* and *variability in time* [PBL05]. The latter refers to variability of artifacts over time in an evolutionary context (e.g., source code artifacts evolve to adapt to changed hardware of a car), which is inevitable for most software systems [PBL05]. In contrast, the former refers to variability in a sense that differing or additional functionality is realized for variants at so-called *variation points* allowing extension with new functionality (e.g., to realize additional driver assistance systems as requested by high-end customers). In the context of this thesis, we are concerned with the analysis of different variants that were created using clone-and-own approaches to adapt their functionality for specific customer requirements. Thus, we use the definition by Pohl et al. [PBL05] for variability in space to define variability in general.

Definition 2.7: Variability

"*Variability* […] is the existence of an artefact in different shapes at the same time." [PBL05]

To talk about the functionality of software systems and their variability (i.e., their common and varying parts), companies often speak of features.

Definition 2.8: Features

"A *feature* is a characteristic or end-user-visible behavior of a software system." [ABK+13]

Based on these definitions, managed reuse can be realized by using specific generative mechanisms to allow easy definition and derivation of product variants from a common platform and additional user-selected features [PBL05, CN01, CE00]. A software family whose artifacts are managed through such a system is also referred to as a software product line.

Definition 2.9: Software Product Line

"A *Software Product Line (SPL)* is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." [CN01]

For example, an SPL for a car would provide the common functionality (e.g., general control software for the engine and all features shared across all variants) and additionally allow configuration (e.g., selection of engine-specific control software when adding a gasoline engine to the car). Arguments for applying an SPL comprise the following aspects [PBL05, CE00, CN01]:

- *Reduction of Development Costs:* Reusing existing shared artifacts and only developing additional features reduces the overall development costs of software systems. However, one has to keep in mind that an upfront investment for the development of the common architecture and functionality is necessary. Nevertheless, the overall cost reduction for all developed systems can outweigh these costs when a sufficient number of variants is included in the SPL.

- *Enhancement of Quality and Reduced Maintenance Effort:* Due to excessive reuse of functionality from the SPL, the corresponding artifacts are tested multiple times. Thus, all derived product variants can benefit from an improved quality. Furthermore, the maintenance effort is largely reduced as fixes can be applied to the SPL's realization artifacts and corresponding variants can easily be regenerated.

- *Reduced Time to Market:* After an initial phase of building the SPL's common artifacts for the first product variants, the subsequent variants can benefit of a reduced time to market due to the increasing level of reuse.

2.3.3. Software Product Line Engineering

The classic *Software Product Line Engineering (SPLE)* process to develop an SPL with its common and varying artifacts is executed in two phases: *domain engineering* and *application engineering* [PBL05]. Both phases are continuous to allow integration of new artifacts in the SPL when new requirements arise. In Figure 2.7, we present the corresponding process according to Pohl et al. [PBL05].

Domain Engineering The *domain engineering* phase is concerned with considering the general design of the created SPL as a whole with corresponding *domain artifacts* to describe its requirements, architecture, components and tests. The sub phases of the domain engineering phase are shown in the upper part of Figure 2.7.

The *product management* phase is concerned with defining a market strategy for the developed software family and to manage the product portfolio with corresponding product road maps. Input to the process are the general goals of the company, which serve as a basis to identify dependencies between the common and varying parts of developed products as well as existing products.

The *domain requirements engineering* phase uses the created product road map to derive specific domain requirements [PBL05]. As the domain engineering is a continuous approach, this sub phase takes existing variability from SPL artifacts into account and identifies common as well as variant-specific requirements for the developed SPL. Output of this phase is a *variability model* (either textual or graphical) clearly grouping similar requirements and highlighting the SPL's variation points. Furthermore, possible variants and identified *constraints* (e.g., to prohibit creation of specific variants) are captured.

The *domain design* phase is concerned with mapping appropriate technical solutions to the identified requirements and realizes a corresponding *reference architecture* for the developed SPL. Engineers have to select an appropriate design for the architecture to support *external variability* (i.e., variability that is visible to the customer [PBL05]) for the created SPL (e.g., the possibility to select certain features). Furthermore, selection of technical options allows to introduce *internal variability* (i.e., variability of the domain artifacts hidden from customers [PBL05]) and enables the necessary variation. Thus, overall this phase is essential to allow mass customization of the product family.

The *domain realization* phase uses the defined reference architecture to implement *reusable software artifacts*. These artifacts comprise the common parts, which are reused for all variants from the product family as well as the varying parts that are only reused for specific variants. By using adequate interfaces and loose coupling between the developed artifacts, the overall reuse can be increased. Output of this phase are individual components that can be later used to derive corresponding variants by combining them.

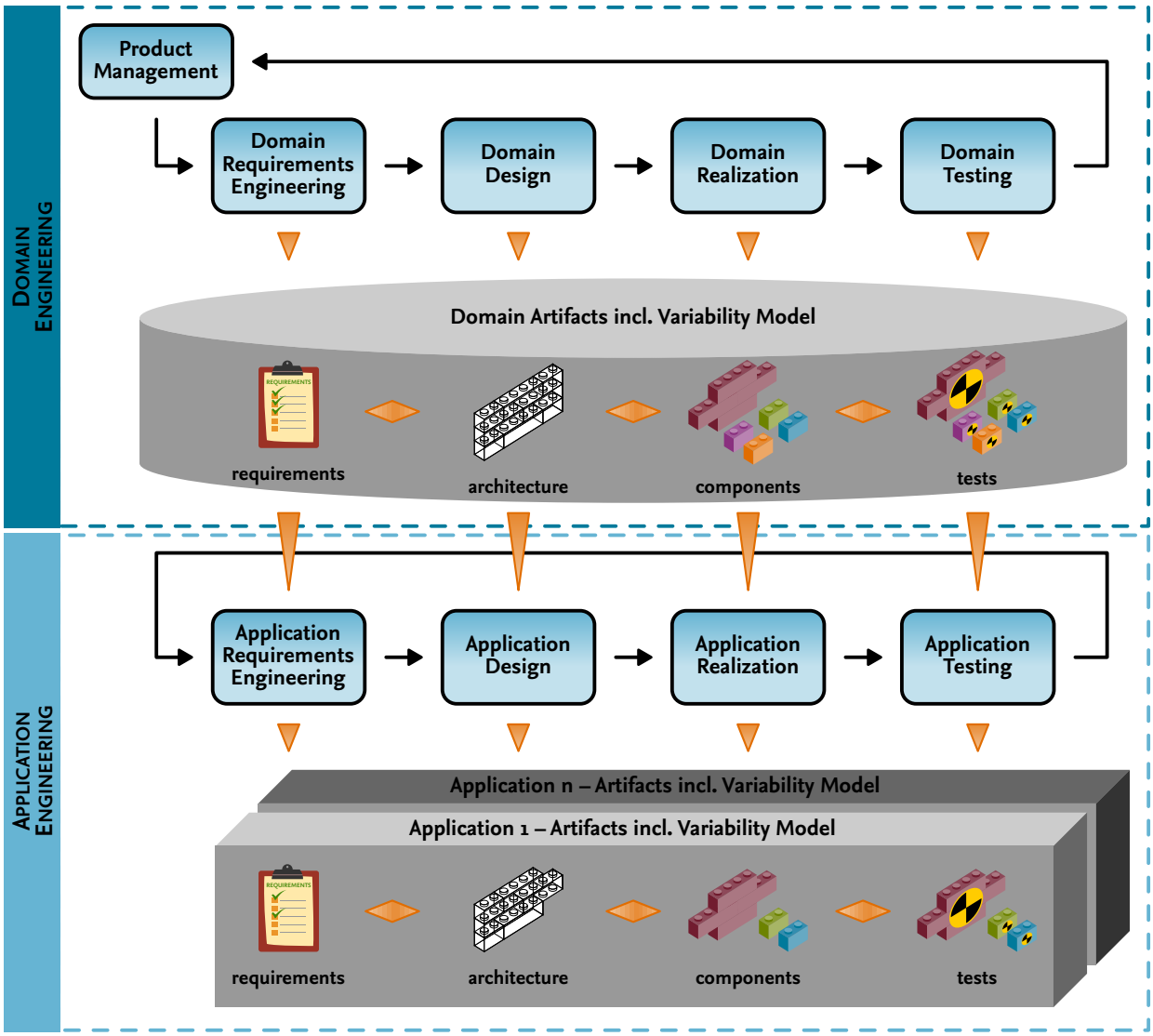


Figure 2.7.: The phases of the classic SPLE process (cf. Pohl et al. [PBL05]).

The *domain testing* phase is concerned with *validation* and *verification* of the developed components to ensure their correct behavior with respect to the specification (i.e., the selected requirements and the architecture). It is important to note that at this point of time no complete application exists and solely tests for single components are executed to ensure their isolated behavior.

As mentioned, the domain engineering process is an iterative process that should be executed iteratively to allow adjustment to changing requirements or the evolution of the SPL.

Application Engineering The *application engineering* phase relies on the results of the domain engineering phase as it is concerned with deriving concrete executable variants from the domain artifacts. The sub phases of the application engineering phase are shown in the lower part of Figure 2.7.

The *application requirements engineering* phase collects the customer's requirements regarding their desired product and, thus, cannot be executed without corresponding input. During this process, the engineers might identify that specific requirements are not yet supported by the existing artifacts. Modification of the requirements and corresponding variability model might be necessary.

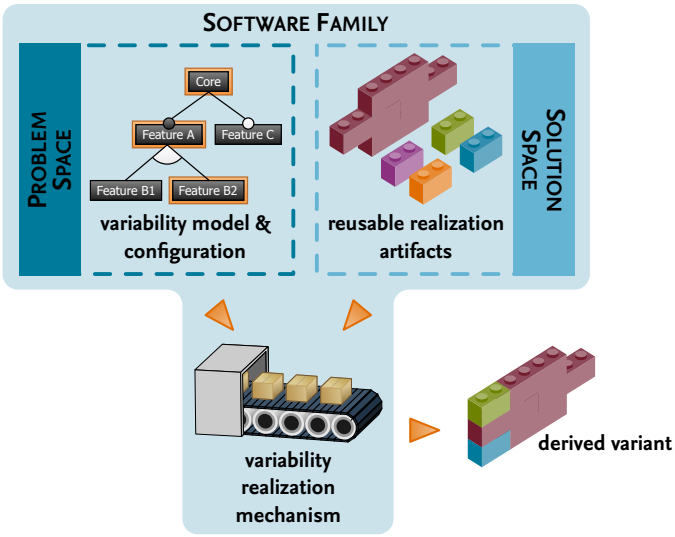


Figure 2.8.: Variability realization mechanism for deriving products from an SPL.

After performing all required changes to the existing SPL artifacts, the complete requirements for the customer's variant can be selected.

The *application design* phase instantiates the reference architecture to derive an application-specific architecture. This phase might involve changes to the reference architecture in order to support the customer requirements. However, due to the previously developed reference architecture this step involves less effort compared to single system engineering and, thus, saves time and money.

The *application realization* phase derives the concrete and fully-functional application using the instantiated architecture and the user-selected features (implemented by reusable software artifacts during the domain realization phase) that might be configured for the customer-specific use case (e.g., through parameterization).

The *application testing* phase is concerned with *validating* and *verifying* the functionality of the derived application against the specification (i.e., the selected requirements). During this phase, test cases from the domain testing phase can be instantiated, which allows for a high reuse. Any errors identified during this phase have the potential to improve the overall quality of the SPL as corresponding fixes can be easily propagated to all other variants.

Problem and Solution Space The described SPLE process is typically divided into the *problem space* and the *solution space* [CE00]. The *problem space* is concerned with describing the concepts of the SPL without concrete realization artifacts by using, for example, requirements and a variability model to describe the configuration space (i.e., all user-selectable options with their dependencies). The variability model supports non-technical experts of the domain (e.g., managers, salesmen, marketing, but also customers) in understanding the developed product and getting an overview of configuration options. The *solution space* is concerned with the concrete realization of the artifacts for all variants in the developed SPL and comprises, for example, source code, design models and corresponding documentation. These artifacts are maintained by technical experts (i.e., the people working on the realization of the SPL) and, thus, are mostly hidden to the customers. Using a suitable *variability realization mechanism* (cf. Section 2.3.5 for a detailed discussion), it is possible to derive concrete variants from the SPL by selecting a valid configuration from the corresponding variability

model. In Figure 2.8, we show how a variability realization mechanism combines the information provided by the problem space (i.e., the variability model) and a user-selected configuration to identify needed realization artifacts from the solution space and to build the desired variant.

2.3.4. Feature Models

A variety of different variability models exist to describe and specify the problem space of SPLs. Prominent examples are *feature models* [KCH+90, CE00], *decision models* [MA02], *Orthogonal Variability Models (OVMs)* [PBL05] and *Variability Specifications (VSpecs)* in the *Common Variability Language (CVL)* [HMO+08]. Using one of these notations, developers can model possible feature combinations and, for example, prohibit specific combinations. As a result, customers are only able to select features and create *configurations* that conform to the modeled variability. This way developers can ensure that the used variability realization mechanism derives only variants supported by the SPL implementation. In this thesis, we employ *feature models* to model the problem space using the notation by Czarnecki et al. [CE00]. A corresponding feature model for the *BCS* case study (cf. Section 2.2) adopted from [LLL+13] and a selected exemplary configuration can be found in Figure 2.9. Each of the boxes represents a feature and, thus, a possible configuration option for the modeled SPL. For space reasons, we used abbreviations in the feature model and refer to Table 2.1 for the corresponding explanations. For the features, we distinguish between:

- *Root Feature:* This feature implicitly has to be selected in all variants derived from the SPL.
- *Intermediate Features:* These features have additional child features below them.
- *Leaf Features:* These features have no child features.

Each feature can either have a distinct *variation type* or be assigned to a *variation group* comprising other related features. Possible variation types are:

- *Mandatory Features:* The feature has to be included in all variants of the SPL, which is shown by a filled circle above the feature. For example, the `PW` feature in Figure 2.9 is mandatory.
- *Optional Features:* The feature does not have to be selected in all variants, which is shown by a hollow circle above the feature. In Figure 2.9, the `Heatable` feature for the *BCS* is optional.

Possible variation groups are:

- *Alternative Groups:* The group features are mutually-exclusive and, thus, only one of the alternative features can be contained in derived variants. Alternative groups are shown by a hollow arc spanning across the connections of all group members to their parent feature. In Figure 2.9, the `ManPW` and `AutoPW` feature represent alternative `PW` systems.
- *Or Groups:* The group features have a logical *Or* relation and, thus, at least one feature has to be included in each variant, but it is possible to select up to all group members. Or groups are shown by a filled arc spanning across the connections of all group members to their parent feature. In Figure 2.9, the six possible `LED` features represent an Or group.
- *And Groups:* This group is implicit if none of the other group types is defined and users can select features obeying the corresponding variability types.

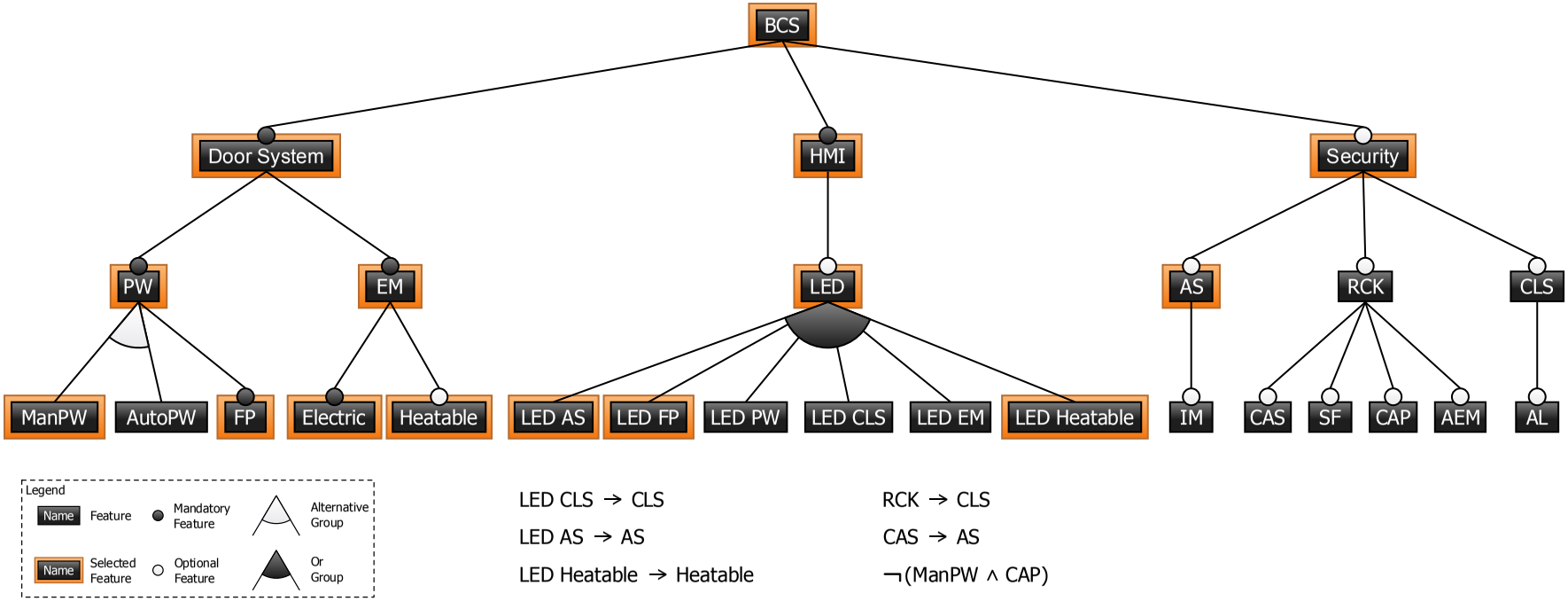


Figure 2.9.: Feature model for the *BCS* case study (cf. Section 2.2) adopted from [LLL+13]. Furthermore the orange boxes show an exemplary configuration selection. For explanations of the used abbreviations, we refer to Table 2.1.

Abbreviation	Name	Abbreviation	Name
AEM	Adjust Exterior Mirror	AL	Automatic Locking
AS	Alarm System	AutoPW	Automatic Power Window
CAS	Control Alarm System	CAP	Control Automatic Power Window
CLS	Central Locking System	EM	Exterior Mirror
FP	Finger Protection	HMI	Human Machine Interface
IM	Interior Monitoring	LED	Light-Emitting Diode
ManPW	Manual Power Window	PW	Power Window
RCK	Remote Control Key	SF	Safety Function

Table 2.1.: Explanations for the feature name abbreviations used in the *BCS* feature model in Figure 2.9.

In addition to these variability relations, it is possible to define so-called *cross-tree constraints*, which can be used to constrain the possible variants. Furthermore, these constraints can span across subtrees of the feature model allowing to express relations that otherwise would not be possible (e.g., that the `LED Heatable` feature requires the `Heatable` feature in Figure 2.9). One possible notation to express these constraints are additional edges between features to show *requires* (i.e., feature `A` requires selection of feature `B`) and *excludes* relations (i.e., the features cannot be selected together) [Bat05, CW07]. However, with these additional edges the resulting feature model can become confusing for users as large numbers of additional and possibly crossing lines are added. Thus, another commonly applied solution to express these constraints is definition of formulas in propositional logic over the features (i.e., each feature is considered as a Boolean variable) [Bat05, CW07]. In Figure 2.9, we show six examples for such cross-tree constraints. For example, the `ManPW` and `CAP` features exclude each other and the `RCK` feature requires the `CLS` feature.

For a valid configuration in a feature model, the user has to select all parent features for any selected feature and obey the variability relations defined for the features. In Figure 2.9, we show an exemplary selection using orange boxes to highlight the selected features. The corresponding variant comprises the `BCS`, `Door System`, `HMI`, `Security`, `PW`, `EM`, `LED`, `AS`, `ManPW`, `FP`, `Electric`, `Heatable`, `LED AS`, `LED FP` and `LED Heatable` features.

Using feature models to represent the problem space of SPLs derived by our SPL migration algorithm for existing variants, we are able to base our created solutions on one of the most common representations for such information. This way, we enable developers familiar with SPLs to easily understand the information and, thus, ease work with correspondingly derived SPLs.

2.3.5. Variability Realization Mechanisms

In academia and industry, a variety of concrete variability realization mechanisms exist for implementing SPLs [SRC+12, Bat04, KAK08, SBB+10]. Depending on the used mechanism the corresponding derivation process as outlined in Figure 2.8 differs. Typically, one can distinguish between *annotative*, *compositional* and *transformational* approaches [SRC+12].

Annotative Variability Realization Mechanisms Annotative variability is often also referred to as *subtractive* or *negative* variability, because it comprises all realization artifacts with their corresponding variability in a single representation and variants are derived by removing unselected parts [SRC+12, KAK08, CA05]. In literature, such unified representations of variability in an SPL are often also referred to as 150% models [SRC+12].

Definition 2.10: 150% Models

A *150% model* comprises all concrete realization artifacts with their possible variability for an SPL and corresponding presence annotations showing the artifacts' mapping to specific features or their containment in specific variants.

Typically, these presence conditions are expressed using additional constructs in a separate language. A prominent example for such an additional language is the C/C++ preprocessor that extends the corresponding compilers and allows to add directives for conditional compilation to the source code. By using the provided constructs, developers can define code that is surrounded by `#ifdef` directives checking for the presence of a specified feature. This way corresponding functionality is

only included in variants where the user selected the matching feature. As we can see, for annotative variability realization mechanisms the solution space is intertwined with the problem space as defined features and annotations are directly linked with their implementation in the realization artifacts. Thus, the variability realization mechanism is able to retrieve all necessary realization artifacts by evaluating these annotations and removing unselected parts from the 150% model. In Figure 2.10, we exemplify the idea of annotative variability by removing bricks from the 150% model to derive an exemplary variant.

A large variety of tools exist to support the annotative variability realization mechanism Examples for industrial tools are BigLever's Gears[6] [Kru08] and pure-system's pure::variants [Beu08]. Furthermore, examples for tools from academia exist in form of the Antenna[7] preprocessor for Java in FeatureIDE[8] [TKB+14, MTS+17], Clafer[9] [BCW11] and FeatureMapper [HKW08].

Compositional Variability Realization Mechanisms Compositional variability is often also referred to as *additive* or *positive* variability, because an existing *SPL core* comprising all realization artifacts shared by all variants is extended with selected additional functionality during variant derivation [Bat04, KAK08, SRC+12]. This SPL core does not necessarily represent a valid variant as, for example, variation points requiring one of multiple mutually exclusive alternatives are only resolved during variant derivation. During variant derivation, the variability realization mechanism adds further code (i.e., selected features) to compose the variant according to the user's configuration. As a result, the link between the concrete realization artifacts and a corresponding variability model is not directly modeled and has to be explicitly defined. Different approaches exist that use, for example, specific naming conventions or explicit mapping declarations. In Figure 2.11, we exemplify the idea of compositional variability and add different bricks to derive an exemplary variant from the SPL core. Prominent examples for compositional variability realization mechanisms are *Feature-Oriented Programming (FOP)* [Bat04] and *Aspect-Oriented Programming (AOP)* [KLM+97].

Transformational Variability Realization Mechanisms In contrast to annotative and compositional variability, transformational variability realization mechanisms are able to *add*, *remove* and *modify* elements in the realization artifacts during the transformation of an existing variant, which serves as basis for the variant derivation [SRC+12]. The atomic *transformation operations* (i.e., add, remove and modify) can be further used to define more complex transformations consisting of multiple operation calls. To define concrete variability for an SPL, developers implement *transformation modules* that encapsulate calls to transformation operations. In principal, developers can choose any variant as a basis for the implemented SPL. For instance, in many cases it can be advantageous to use a variant that comprises the largest overlap between the SPL's variants to reduce size of developed transformation modules. Similar to compositional variability, additional mappings between the SPL's variability model and corresponding realization artifacts are necessary to allow derivation of variants using transformational variability realization mechanisms. In Figure 2.12, we exemplify the idea of transformational variability and add, remove and modify bricks to derive an exemplary variant from the SPL core.

[6] http://www.biglever.com/solution/product.html
[7] http://antenna.sourceforge.net/wtkpreprocess.php
[8] https://featureide.github.io/
[9] http://www.clafer.org/

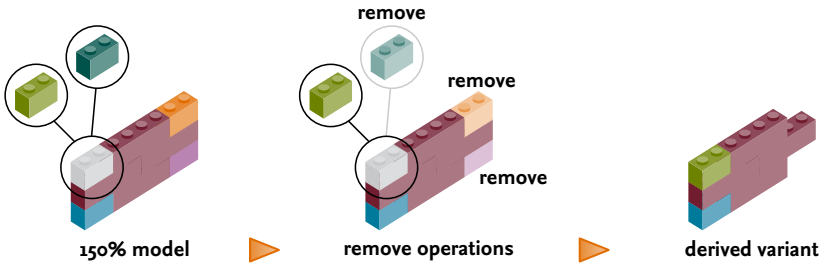


Figure 2.10.: Annotative variability realization mechanisms remove concrete realization artifacts, which were not selected by an input configuration, from the 150% model to derive corresponding variants.

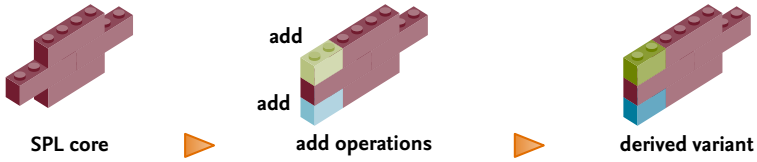


Figure 2.11.: Compositional variability realization mechanisms add concrete realization artifacts, which were selected by an input configuration, to an SPL core to derive corresponding variants.

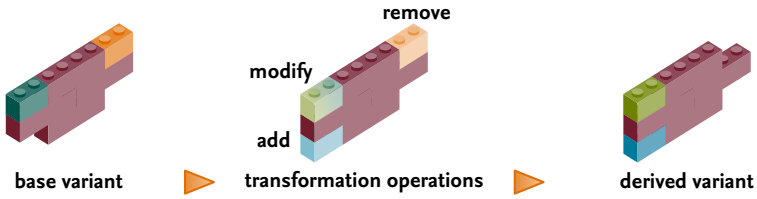


Figure 2.12.: Transformational variability realization mechanisms apply add, remove or modify operations to transform an existing base variant to derive new variants based on selected input configurations.

In general, it is possible to use different approaches to implement transformational variability realization mechanisms. For example, it is possible to apply *general purpose transformation languages* to execute the necessary transformations between variants. However, as such general purpose transformation languages use generic transformation operations and, thus, lack a syntax similar to the originally used language (e.g., Java), *domain specific transformation languages* exist to allow easier definition and comprehension of transformations [RW11]. Another approach, specifically targeting this problem, is *delta modeling* – also referred to as *Delta-Oriented Programming (DOP)* [SBB+10, SD10].

2.3.6. Delta Modeling

Delta modeling relies on *delta languages* providing transformation operations that are specifically tailored towards the used programming language as they reuse corresponding vocabulary in the operation names [SBB+10, SD10]. These operations are referred to as *delta operations* and allow to *add*, *remove* or *modify* elements from existing variants. Thus, in contrast to general purpose transformation language, delta modeling eases understanding of developed functionality by experts, because applied delta operations clearly allow to understand executed actions (e.g., DeltaJava [SBB+10, KHS+14] contains operations to add, remove and modify attributes of classes). Based on the atomic add, remove and modify operations it is possible to define more complex operations (e.g., allowing to add a transition to a statechart that automatically sets the transition's source and target states).

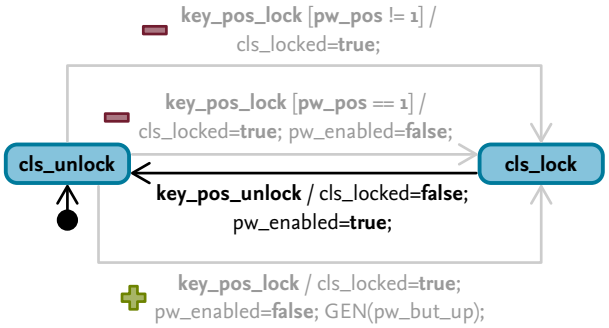


Figure 2.13.: Exemplary delta operations executed for our running example to transform the CLS variant with ManPW support (cf. Figure 2.5a) to the CLS variant with AutoPW support (cf. Figure 2.5b).

Using such delta languages developers can create transformations of existing variants by defining *delta modules* storing *delta operation calls* triggering the necessary operations.

By selecting a set of delta modules, variants are executed from an existing variant by executing the stored delta operation calls in sequential order. Dependencies between delta modules can exist as, for example, one delta module might require that another delta module adds a certain model element prior to modifying it. Thus, delta modules can define *application order constraints* allowing the variability realization mechanism to determine the required execution order.

To ease derivation of variants from delta-oriented SPL, it is possible to define *mappings* between the features of a feature model and corresponding delta modules implementing them. Thus, these mappings constitute a direct mapping between problem and solution space and allow selection of variant *configurations* directly from the feature model.

In Figure 2.13, we show the exemplary derivation of the CLS variant supporting an AutoPW (i.e., the variant in Figure 2.5b) from the CLS variant supporting an ManPW (i.e., the variant in Figure 2.5a) in our running example. For this transformation, we have to execute the following delta operations: First, we have to remove the two transitions in the upper part of the figure (i.e., the two transitions used to distinguish between the positions of the PW) – shown by the red minus symbols. Second, we have to add the new transition in the lower part of the figure (i.e., the transition used to automatically generate the command to close the window) – shown by the green addition symbol. An additional example (not shown in Figure 2.13) for a modification delta operation could be the renaming of the transformed variant to reflect the changed functionality (e.g., a renaming from CLS ManPW to CLS AutoPW).

DeltaEcore In this thesis, we employ DeltaEcore[10] [SSA14, Sei15] to migrate existing product variants to a delta-oriented SPL. DeltaEcore is an extensible framework that allows creation and application of delta languages for EMOF-based languages. In Figure 2.14, we give an overview of the DeltaEcore framework and its provided tooling. The framework provides a *common base delta language*, which allows easy definition of delta operations for language-specific meta-models. While *set* and *unset* operations assign a new value to a *single-valued reference* or replace the current reference with the default value, *add* and *remove* operations add a reference to a *many-valued reference* or remove it, respectively. In addition, an operation to modify object *attributes* exists. To this end, the user can define custom *delta dialects* storing definitions for the delta operations required to transform the

[10] http://www.deltaecore.org/

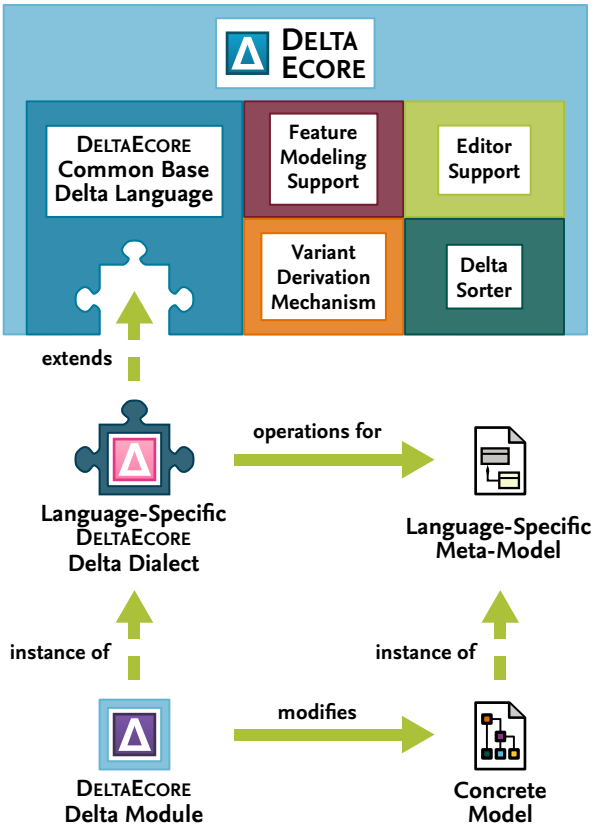


Figure 2.14.: Overview of the DeltaEcore framework and its support for custom delta languages.

desired references or attributes in the language-specific meta-models. Together with the common base delta language, the user-specific delta dialect forms the delta language that can be used to define concrete delta modules transforming existing models. As DeltaEcore allows to generate fully functional delta languages from corresponding delta dialects, we use both terms as synonyms.

To support developers during development of an SPL, the DeltaEcore framework provides feature modeling support and editor support for delta modules using any of the created delta languages. In addition, DeltaEcore provides facilities to derive variants by applying delta modules that were automatically sorted according to their user-specified dependencies.

We employ delta modeling for our SPL migration, because it supports different software product line adoption models including the extraction of necessary artifacts from existing implementations [SD10]. Furthermore, the possibility of a highly modular design in delta-oriented SPLs allow for high flexibility and enable us to migrate existing functionality to features with corresponding delta modules. This way, we are not only able to dissect existing variants into identified features, but also enable developers to easily extend the derived SPLs with additional functionality.

2.3.7. Software Product Line Adoption Models

In Section 2.3.3, we have presented the general SPLE process to develop and maintain SPLs. However, depending on a company's requirements or situation, different adoption models can be used to support the current goals [Kru01]. According to Krueger [Kru01], they can be categorized into *proactive*, *reactive* and *extractive* models.

Proactive Adoption Model The proactive adoption model is most appropriate in situations where a company already has a deep understanding of the targeted domain and its requirements. The overall goal is to develop an SPL that supports the identified products and, thus, addresses a wide range of potential future customers.

In Figure 2.15, we visualize the proactive adoption model. For this model, Krueger [Kru01] speaks of an adoption strategy similar to the waterfall model for single systems and the upper part of the figure basically represents the domain engineering from the classic SPLE process according to Pohl et al. [PBL05]. Similarly, the lower part (i.e., the variability realization mechanism) represents the application engineering described by Pohl et al. [PBL05] (cf. Section 2.3.3) to derive concrete variants from the artifacts developed during domain engineering.

Reactive Adoption Model The reactive adoption model is relevant in situations where additional requirements for new products become apparent and the existing SPL might need adaptation.

In Figure 2.16, we visualize the reactive adoption model. For this model, Krueger [Kru01] assumes that already an SPL realization exists and the company identified a new product variant that should be provided to a customer. In this case, the new requirements for this product have to be analyzed and compared with the existing requirements of the SPL. Ideally, the SPL already supports the desired product variant, in which case it can be easily derived using the applied variability realization mechanism and the existing realization artifacts. However, in certain cases the developers might identify that the requested functionality is not yet supported and additional artifacts have to be implemented to support it. For the necessary adoption, the developers have to execute an iteration of the classic SPLE approach by Pohl et al. [PBL05] (cf. Section 2.3.3) to ensure a controlled extension of the developed SPL. As shown in Figure 2.16, the reactive adoption model is an iterative process that is used for the evolution of the SPL whenever new products are requested.

Extractive Adoption Model The extractive adoption model is a strategy to migrate existing product variants to an SPL. It allows to reuse implemented functionality from these existing product variants and, thus, enables companies to reduce overall implementation effort during the adoption of SPL techniques.

In Figure 2.17, we visualize the extractive adoption model. For this model, Krueger [Kru01] states that it is necessary to identify the commonalities and differences between the existing products. Afterwards, users should use a copy of the common software parts as a basis for the developed SPL and create feature declarations to model the variability and implement them. Using the created SPL, it is possible to derive all variants that served as basis for its realization. Overall, the descriptions of the different steps in [Kru01] are not detailed enough to serve as concrete guidelines. For example, no concrete details on the process for identifying commonalities and differences is given and it must be assumed that this step has to be executed completely manually without tool support.

Overall, the described adoption models do not have to be seen as mutually exclusive strategies [Kru02]. For example, the proactive or extractive adoption models can both be used to realize an initial SPL that can be further extended using the reactive adoption model when identifying new requirements for additional functionality. In this thesis, we follow an extractive adoption model as we analyze existing variants that previously were developed using a clone-and-own approach.

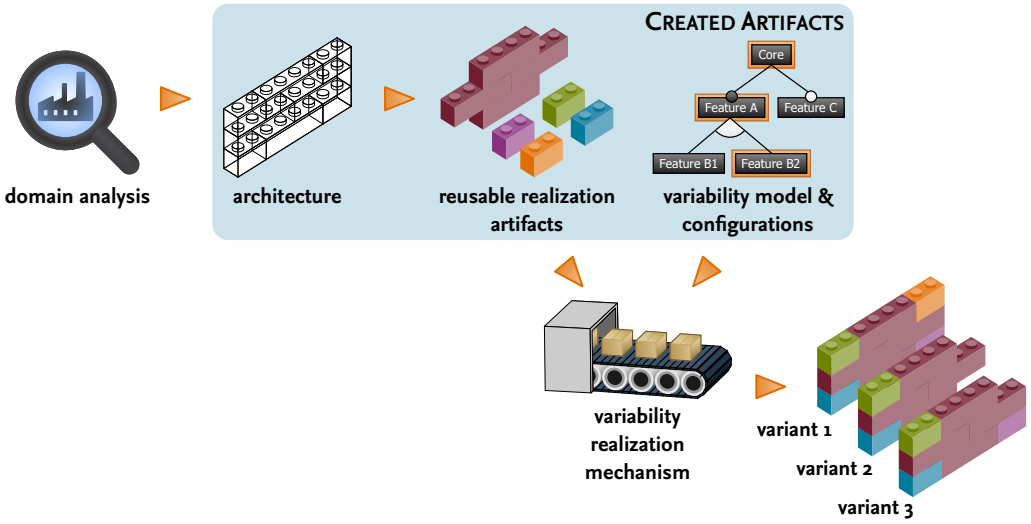


Figure 2.15.: The proactive adoption model follows the classic SPLE process introduced by Pohl et al. [PBL05] (cf. Section 2.3.3) to develop an SPL for a specific software system (cf. Krueger [Kru01]).

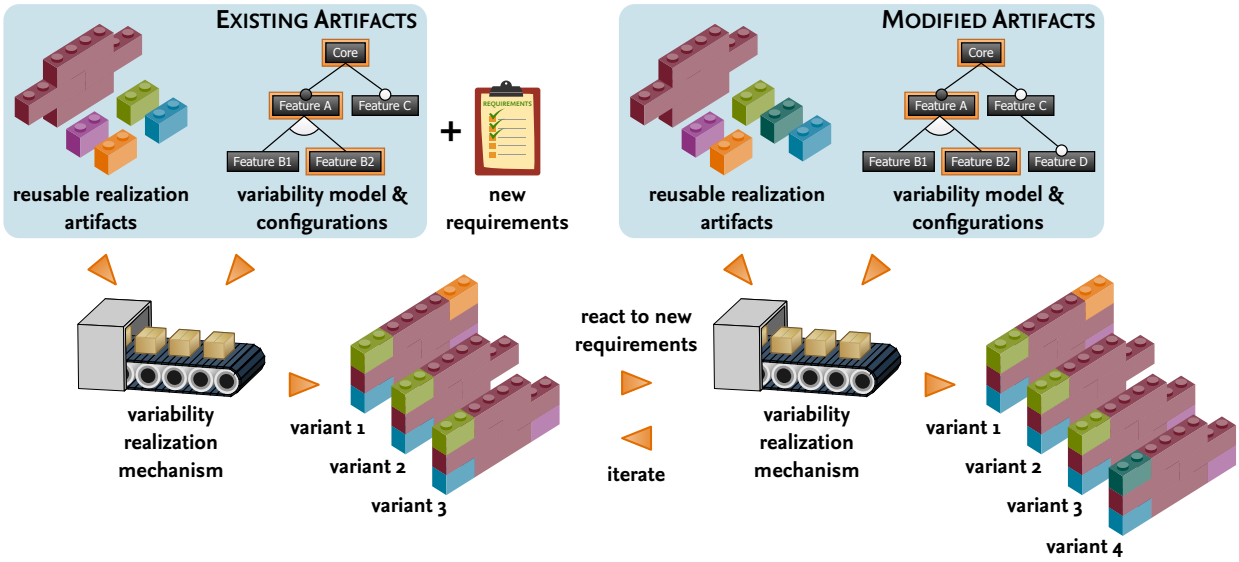


Figure 2.16.: The reactive adoption model iteratively extends an existing SPL when new requirements regarding additional functionality of the developed SPL become apparent (cf. Krueger [Kru01]).

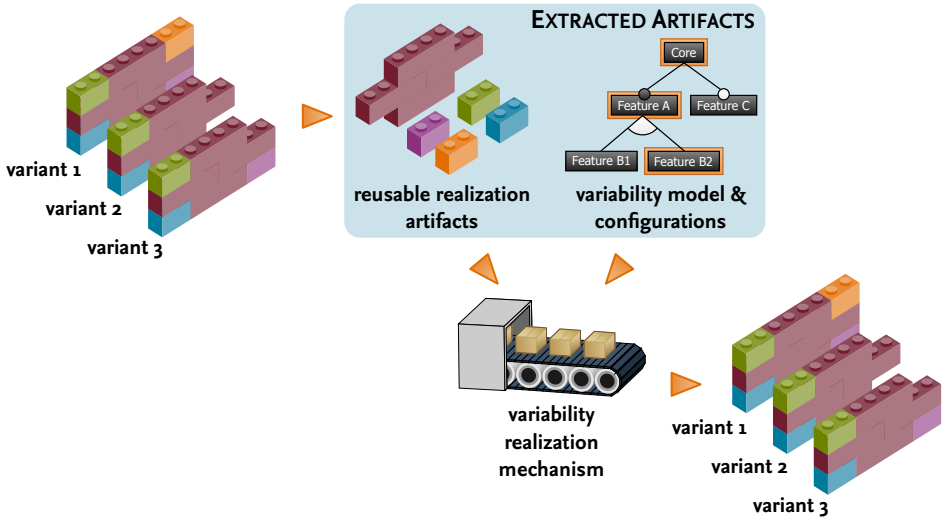


Figure 2.17.: The extractive adoption model uses existing variants of a software system (e.g., previously developed using a clone-and-own approach) to extract their common and varying parts. Using the extracted artifacts the analyzed variants are migrated to a corresponding SPL (cf. Krueger [Kru01]).

Part II.

Custom-Tailored Product Line Extraction

3 Detecting Clusters and Removing Outliers

The contents of this chapter are largely based on the work published in [WBC+18].

Summary *Understanding high-level relations between models in clone-and-own scenarios (e.g., which variant was the basis for another variant) can be a tedious task as often no documentation on the variants' relationships exists. Especially, in cases, where a competitors company was acquired and corresponding products were integrated in the existing portfolio, developers might be faced with a large number of unknown variants. Such unknown relations can have negative effects on low-level analysis of existing variability as comparing unrelated variants can result in unexpected relations confusing developers. Also analyzing subsets of variants might improve the results depending on the use case. For example, developers can neglect information irrelevant for their current task and focus on details specifically showing the problem at hand (e.g., variability information specific to a certain model feature). To overcome these problems, we describe our algorithms to analyze large sets of models on a high level of abstraction to identify their relations for a cluster and outlier detection. By providing a visualization of the results, we allow guidance of developers during the exploration of the variants.*

Transitioning a set of existing model variants to an SPL realization requires a detailed understanding of the variability relations between the variants. However, in case of uncontrolled clone-and-own approaches, these relations between model variants are often not known as, for example, only incomplete documentation for their development history (i.e., which variant served as parent for other variants) exists if at all. As a result, developers first have to get a rough overview of the high-level relations between the model variants. However, identifying which products are worth comparing in detail becomes a tedious task as an extensive analysis of variability relations has to be executed due to the lack of documentation. For instance, such detailed relations might not be documented after acquiring a competitors company and integrating the corresponding products in the own portfolio. We refer to variability relations on this high level of abstraction as follows.

Definition 3.1: Coarse-Grained Variability Relations

Coarse-grained variability relations describe the variability between complete model variants on a high level of abstraction. They only show to what extent the variants are related (e.g., two model variants mv_1 and mv_2 are 80% similar) and do not give details on the corresponding low-level variability (e.g., the variants differ in a set of specific model elements).

As a result, coarse-grained variability relations only show that specific variants have a relation and are worth to be considered for a detailed variability analysis. Without such prior knowledge, the detailed analysis of low-level variability (e.g., which transitions of statechart variants are alternative to each other) can result in unexpected results as completely unrelated variants (e.g., from a

different product line) have a negative influence on the identified variability. For instance, parts of the models are recognized as optional to the family of variants, which otherwise would be regarded as mandatory when ignoring the unrelated variants. We refer to such unrelated variants as follows.

Definition 3.2: Outlier Variants

Outlier variants (or for short *outliers*) represent variants that are completely unrelated to a set of other highly related variants.

Such outliers might exist in cases, where a developer left a company and knowledge about the developed variants is lost or when variants from a different department or company (e.g., after acquiring it) are integrated in an existing pool of variants. When comparing such an outlier variant with a set of highly related variants, the results can contain variability relations that are completely unexpected by the experts (e.g., parts are recognized with an unexpected variability).

Furthermore, it might be desirable to execute detailed analysis of variability relations between subsets of variants only. This allows to provide developers with viewpoints on variability relations that are specifically targeted on their current task (e.g., maintenance of a specific feature). We refer to such subsets of related variants as follows.

Definition 3.3: Clusters of Related Variants

Clusters of related variants (or for short *clusters*) are formed by variants whose implementations are highly similar to each other and, thus, exhibit a high *inter-cluster* similarity. Depending on the analyzed data set, they might have a considerably lower *intra-cluster* similarity allowing distinction against other clusters.

An example for a potential cluster are variants that implement mostly the same features and differ only in small portions (e.g., when implementing a small number of additional features). Considering only such clusters of variants for a detailed variability analysis on a low level of abstraction can be advantageous. For instance, the number of compared variants and, as a result, the corresponding comparison times can be reduced. Furthermore, concentrating only on clusters of variants evolving around a set of common features can allow viewpoints on complex data that allow users to ignore information irrelevant for specific tasks. For instance, fixing a bug in a set of variants comprising a specific feature can be eased when hiding unrelated implementation details.

In Figure 3.1, we give an overview of our CoreVID[1] approach to identify coarse-grained variability relations. The approach builds upon the Samos[2] framework, which was originally designed by Babur et al. [Bab16, BCB16, BCV+16, BC17, BCB18] to execute statistical analysis of models and meta-models to identify clusters of related models or meta-models. The Samos framework for model clustering is inspired by *Information Retrieval (IR)* [MRS08] and uses techniques for document indexing (e.g., as in search engines) to make documents comparable. In contrast to detailed model comparison algorithms (e.g., complex graph-matching algorithms), the approach by Babur et al. is more interested in approximate relations to make large sets of models comparable on a high level of abstraction, which would not be possible with such detailed algorithms due to the high complexity and linked costs for in depth analysis of the models. Thus, we decided to use this framework for

[1] **CO**arse-Grained **R**elations **E**xtraction for **V**ariability **ID**entification
[2] **S**tatistical **A**nalysis of **MO**del**S**

our approach as it provides the necessary tooling and algorithms to analyze large sets of models and to determine corresponding outliers and clusters. The CoreVID approach utilizes the corresponding statistical facilities and extends them with problem-specific solutions (e.g., specific handling of identifier names in models) to identify clusters and outliers in a set of input models. Overall, the goal of the CoreVID approach is to support developers in regaining an overview of large sets of model implementations with unclear relations (e.g., in clone-and-own scenarios) prior to selecting subsets of these models for a detailed analysis of their low-level variability. During the statistical analyzes of the input models, the CoreVID approach executes the following three phases:

- *Extracting Relevant Model Details:* During this phase, the CoreVID approach utilizes IR techniques provided by the Samos framework to extract so-called IR-features (i.e., abstracted details) from the input models that are relevant for the corresponding comparisons.
- *Comparing the Extracted Model Details:* Next, the CoreVID approach applies different *Natural Language Processing (NLP)* techniques, such as tokenization and weighting, provided by the Samos framework to preprocess the extracted information and compare it.
- *Clustering Models:* Based on the resulting *Vector Space Model (VSM)* – i.e., a representation of the compared information, the CoreVID approach applies distance measures to calculate the similarity of the compared models and assigns them to clusters. The results can be visualized in dendrograms and allow developers to identify outliers or to select clusters for further analyzes with low-level variability mining approaches (cf. Chapter 4).

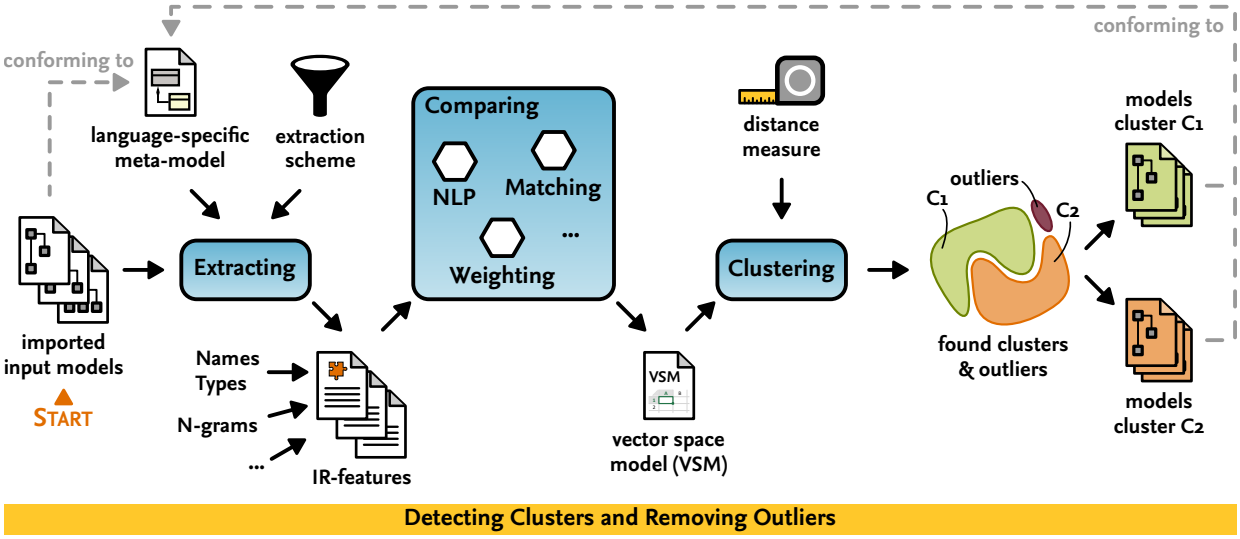


Figure 3.1.: Workflow of the CoreVID approach to detect outliers and clusters in a set of model variants.

Chapter Outline Our goal in this chapter is to focus on describing how we apply the Samos framework to support developers in identifying clusters and outliers in large sets of model variants. While a variety of approaches and algorithms exists to solve the problem at hand (e.g., different clustering algorithms and NLP processing techniques exist) [JD88, MRS08], we build our CoreVID approach on the previously evaluated capabilities of the Samos framework. A detailed discussion of different approaches (e.g., clustering algorithms) and their advantages over each other is outside of the scope

of this thesis. We rather concentrate on showing the general feasibility of the approach and how to apply the Samos framework for our problem at hand.

In the following sections, we explain the details of each phase by applying the described techniques to our running example:

- *Section 3.1:* To allow for a sensible comparison of models on a high level of abstraction, we first need to extract relevant model details. In this section, we explain on how to use IR techniques to translate models into typed n-grams enabling an efficient high-level comparison while preserving certain structural details (e.g., the model hierarchy) to ensure meaningful results.
- *Section 3.2:* Based on the extracted information, we apply NLP techniques provided by the Samos framework to preprocess the corresponding data and compare it.
- *Section 3.3:* These results are used to apply clustering algorithms and to assign the analyzed model variants to different clusters or identify them as outliers.
- *Section 3.4:* This section provides a comparison with related work in the area of statistical model analysis with the goal of identifying high-level relations.

3.1. Extracting Language-Specific Typed N-grams

In an initial step, it is necessary to extract relevant details from the analyzed models (e.g., the states contained in statecharts together with outgoing transitions). We refer to these details as IR-Features.

Definition 3.4: IR-Features

IR-features represent terms contained in analyzed documents (i.e., models in our case). Normally, these IR-features are simply referred to as "features" in the IR domain. However, to make a clear distinction between features in the context of SPLs (cf. Definition 2.8), we refer to them as IR-features in the context of this thesis.

IR-features can be used to create an index of all analyzed documents with clear links between IR-features and containing documents. Selecting the right level of abstraction for these IR-features is essential for the success of the executed clustering [MRS08]. This is necessary to enable efficient and scalable comparison and clustering of large sets of models without loosing information relevant to achieve meaningful results. On the one hand, they represent the "vocabulary" used in the compared documents and, thus, need to be as detailed as possible to allow sensible comparisons. On the other hand, these IR-features have to abstract from the models to allow comparison in reasonable time.

Extracting such IR-features from model variants is supported by the Samos framework through an interface for *extraction schemes*. The interface builds upon the EMF *Application Programming Interface (API)* and allows to implement language-specific extraction of IR-features for Ecore-based meta-models in form of typed n-grams.

Definition 3.5: Typed N-grams

Typed n-grams allow to encode structural details from the analyzed models (i.e., the graph-like structure of Ecore models) together with the types of analyzed model elements (i.e., the name of the corresponding meta-model class) and their names [BC17].

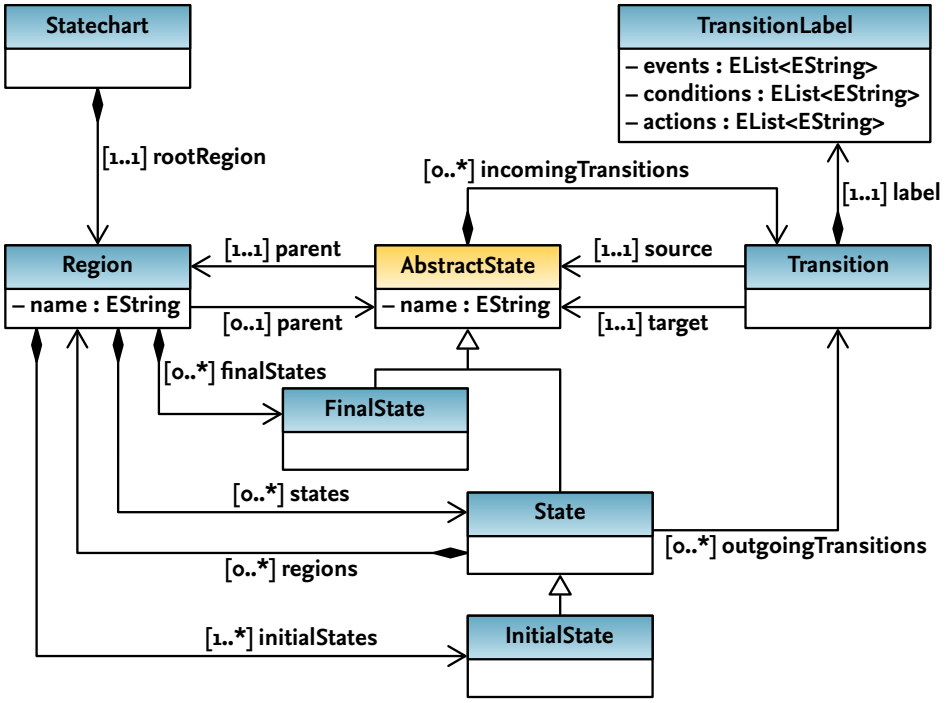


Figure 3.2.: Simplified meta-model for statecharts.

With an increasing n additional details on the models' structure can be analyzed and not only the existence of elements is considered, but also whether they coexist with the same structures in other models. For example, *typed unigrams* (i.e., $n = 1$) consider each element in isolation (e.g., the states and transitions in statecharts), while *typed bigrams* (i.e., $n = 2$) already consider details about the relations between the elements (e.g., that a certain state has a specific outgoing transition).

Finding the right n for the extraction of typed n-grams depends on different factors. In their work, Babur et al. evaluate different sizes for the extract typed n-grams [BC17]. The results show that, while increasing the n might improve the accuracy of the results (e.g., as additional structural information is encoded), it increases the complexity and, thus, the runtime of the algorithm [BC17]. Furthermore, a higher n does not automatically result in improved results, but can also have no effect or even decrease the quality in certain cases [BC17]. As the evaluation by Babur et al. suggests that typed bigrams perform well regarding the complexity in most cases, while improving the accuracy on average [BC17], we decided to employ $n = 2$ for the extraction in our Corevid approach. However, in case experiments with the existing data show that applying n-grams with a higher n would improve the results, the Corevid approach would allow to use them. Using the Samos framework's interface for extraction schemes it is easily possible to integrate this additional information in the process.

In Figure 3.2, we show a simplified meta-model for statecharts, which we use to explain how the Corevid approach extracts exemplary typed bigrams for the two CLS statechart variants with differing PWs in Figure 2.5. In this case, our Corevid statechart extraction scheme considers the following details of the meta-model:

- `Region`, `State`, `InitialState`, `FinalState`: including their names and types
- `Transitions`: with their `events`, `conditions` and `actions` in the `TransitionLabel`

These details are encoded in typed bigrams using the following relations:

- `Region` → `InitialState` via the containment relation `initialStates`
- `Region` → `State` via the containment relation `states`
- `Region` → `FinalState` via the containment relation `finalStates`
- { `InitialState`, `State` } → { `Event`, `Condition`, `Action` } via the `TransitionLabel` of the `outgoingTransitions` of the `State` class
- { `Event`, `Condition`, `Action` } → { `InitialState`, `State`, `FinalState` } via the `targetState` of the `Transition` class

For the last two relations, we summarized the possible elements of the relations as sets. Actual typed bigrams using these relations can only contain one of the elements from each of the corresponding sets (e.g., `State` → `Event`). Based on these relations, the Corevid approach is able to extract corresponding types bigrams. Examples for the CLS ManPW variant in Figure 2.5a are:

```
[InitialState, cls_unlock] - [Event, key_pos_lock],
[Event, key_pos_lock] - [State, cls_lock],
[InitialState, cls_unlock] - [Condition, pw_pos != 1],
[Condition, pw_pos != 1] - [State, cls_lock]
[InitialState, cls_unlock] - [Action, cls_locked=true;],
[Action, cls_locked=true;] - [State, cls_lock]
```

Due to the structure of the extracted typed n-grams, the subsequent clustering approach might be vulnerable to extreme scenarios. For instance, when renaming all identifiers in a copy of a model variant, the Corevid approach would most likely not detect the relation to the original variant. Main reason is that, although, the Corevid approach considers partial information on the model structure (i.e., in the size of the selected n), the clustering mostly relies on the names of model elements encoded in the typed n-grams. More complex techniques, fully relying on the structural information of models (e.g., graph-matching algorithms), might be able to detect such scenarios as they would be able to identify related graph patterns. However, these algorithms often follow a different goal as they are more concerned in identifying exact matches between models on a much more fine-grained level, while the Corevid approach is more interested in approximate relations on a high level. Thus, the Corevid approach has its advantages in classic clone-and-own scenarios, where functionality is mostly extended and models are not completely renamed. Here, the abstraction from the complete model structure in form of typed n-grams largely reduces the complexity of comparing models, while still providing enough information to identify the general relations between them (i.e., possible clusters and outliers). This is of crucial importance in the main application scenarios of the algorithm, where large sets of models might hinder application of complex and costly techniques, such as graph matching. In fact, the Corevid approach could further be extended with additional NLP processing for the comparison of n-grams by using the facilities provided in the Samos framework. By defining additional synonym checking, developers might use their knowledge about the executed renamings to account for the limitation of the abstract view on

the compared model and to detect such scenarios. In addition, we argue that scenarios with completely renamed model elements only rarely occur in realistic industrial settings. Thus, the chance of encountering such problems is almost neglectable.

Overall, the extraction of n-grams can be easily adapted for meta-models of other languages as developers only have to define language-specific extraction schemes in the Samos framework. Using the corresponding interface together with the EMF API makes it particularly easy to implement such extraction schemes and only requires a small amount of user-specific source code. The remaining clustering steps are completely language-agnostic in a sense that they are executed on the extracted bigrams. Thus, the corresponding analysis steps are completely automatic and require no further manual input by the user.

3.2. Applying Natural Language Processing to Compare the Typed N-grams

To further process the typed bigrams extracted by the user-defined extraction scheme of the Corevid approach, we configure the Samos framework to execute the actual comparisons between the models. For this purpose the Samos framework provides a large variety of IR and NLP techniques that can be individually selected depending on the problem at hand. The following list gives an overview of the techniques provided by the Samos framework to execute these calculations. For a complete list of all available techniques, we refer to [BCB16, BC17].

- *Tokenization:* This option allows to split words into tokens (i.e., substrings of the original string) and remove certain parts (e.g., punctuation or whitespace) [MRS08]. For example, the words of a sentence represent tokens [MRS08].
- *Stop Word Removal:* Stop words are filtered out by NLP techniques to remove irrelevant tokens that add little or no meaning to the analyzed texts [MRS08]. For example, in an English sentence the articles *the*, *a* and *an* can be regarded as such stop words, because they do not give additional information about the sentence's subject. However, depending on the context of the analysis it is possible to select any word as a stop word (e.g., to remove all numbers).
- *Normalization:* Allows to account for marginal differences by canonicalizing the tokens of a text [MRS08]. For example, normalization could be applied to account for different spelling forms in American and British English, such as *formalization* vs. *formalisation* or *color* vs. *colour*.
- *Stemming:* This option reduces words to their common stem, which eases checking for synonyms [MRS08]. For example, *located*, *location* and *locations* would be reduced to *locat* and, thus can be regarded as synonyms.
- *Zones:* Allows to separate important parts of a document from the rest of the text and assign higher weightings to them [MRS08]. For example, the *title* or an *abstract* can be regarded as specific zones for a paper as they have a different meaning than the regular contents.
- *Weighting Schemes:* Weighting allows to assign different relevance to specific tokens or to consider partial similarity [MRS08]. For example, the *Inverse Document Frequency (IDF)* of tokens in a document is a common way to increase the discriminative impact of rarely occurring words [MRS08].

- *Semantical Relatedness Detection:* Semantic relatedness detection allows to expand the search space as queries are not limited to exact matches and possible additional relations are taken into account [MRS08]. For example, it is possible to detect not only synonyms of words (e.g., *house* vs. *building*), but also to detect other relations, such as hyponyms. Hyponyms are words that share a common ancestor in a logical and semantical sense. For example, an *eagle* and an *elephant* are both *animals*, but not both *mammals*. A commonly used library to detect such semantic relations is Wordnet[3], which provides a database of English words and allows querying of semantic and lexical relations [Mil95, MRS08].

To this end, we enabled the Levenshtein Distance algorithm [Lev66] to identify the edit distance between identifiers to account for potential typos and stemming to remove affixes from compared identifiers. In addition, we turned off specific settings that are not required for our approach. For example, costly checks for semantic relatedness (e.g., using Wordnet) are not required as the vocabulary space for analyzed models should be limited to identifiers that were selected by developers based on similar requirements of a company. However, in certain cases it might make sense to activate these additional techniques. For example, when inspecting the different variants based on the generated dendrograms reveals that the analyzed models are much more related than shown (e.g., because different developers decided to use synonymous identifiers for otherwise equivalent concepts). We also disabled advanced weighting (e.g., through the IDF) and only included simple weighting schemes based on the types of model elements considered in the extracted bigrams. In other words, when comparing `[State, cls_lock]` with `[Region, cls_lock]`, we still consider them to be 50% similar as their names are equal despite the different types.

Furthermore, we extended the Samos framework with additional NLP capabilities to handle commonly used identifiers in models or source code. For example, we added algorithms that allow tokenization of identifiers in *snake case* (e.g., `token1_token2_...`), which allows improved handling of such identifiers (e.g., the name of the `cls_unlock` state in Figure 2.5a). In addition, we applied a trick to enable sensible comparisons of complete source code expressions or statements (e.g., `cls_locked!=true;` in Figure 2.5a) without applying complex techniques to parse them. Here, we extended the stop word list with operators, primitives (e.g., boolean primitives `true` and `false`), parentheses, semicolons and brackets, which reduces the parsed elements to the identifiers of such statements. As a result, the NLP techniques provided by the Samos framework are able to detect similarities between such identifiers (e.g., `cls_unlock` vs. `cls_lock`) or expressions (e.g., `cls_locked=true;` vs. `cls_locked=false;`).

After applying the selected processing techniques to the extracted n-grams, the result are translated to so-called vector space models.

Definition 3.6: Vector Space Models

Vector Space Models (VSMs) are $m \times n$ matrices representing m documents with a vocabulary of size n [MRS08].

Each document in a VSM is represented by a single vector (also called *term incidence vector* [MRS08]) showing which elements of the overall vocabulary across all analyzed documents occur in the specific document (and which frequency they have). The *term incidence matrix* formed by a VSM does not

[3] `https://wordnet.princeton.edu/`

contain negative values as the lowest number of occurrences for a word in the vocabulary is 0. Furthermore, each word in the vocabulary occurs at least in one of the documents, otherwise, it would not be included at all. In our case, the m input models to the Corevid approach represent the documents of the VSM and the extracted and further processed n typed bigrams form the vocabulary of the $m \times n$ term frequency matrix.

3.3. Clustering the Typed N-grams Using a Vector Space Model

Based on VSMs it is possible to apply additional statistical methods to calculate the similarity of the vectors and to perform clustering.

Calculating the Distance between Models To calculate relations between compared models, the Samos framework provides different distance measures.

Definition 3.7: Distance Measures

Distance Measures allow to calculate the distance (e.g., the dissimilarity) between vectors of documents [MRS08].

These distances can later be used during clustering to assign the identify clusters of similar documents. Distance measures provided by the Samos framework include the following algorithms:

- *Manhattan Distance:* The *Manhattan Distance* for two vectors p and q of n dimensions is:

$$manhattan(p,q) = \sum_{i=1}^{n} |p_i - q_i|$$

  Thus, it is basically the sum of vertical and horizontal segments between two points in a coordinate system.

- *Euclidean Distance:* The *Euclidean Distance* for two vectors p and q of n dimensions is:

$$euclideanDistance(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

  Thus, it represents the direct distance between two points in a coordinate system.

- *Cosine Dissimilarity:* While *Manhattan Distance* and *Euclidean Distance* are able to calculate the distance between two vectors, they are not normalized, which makes comparison of vector similarities more complex. In addition, these two measures are not able to account for documents of different length. As a result, the specific tokens might only occur more often in a document, because it is longer and, thus, similarities between documents are not correctly detected. In contrast, *Cosine Dissimilarity* (i.e., $1 - cosineSimilarity(p,q)$) for two vectors p and q of n dimensions is length normalized in the interval $[0..1]$ and can correct a higher frequency of tokens due to differing document sizes:

$$cosineDissimilarity(p,q) = 1 - \frac{p \cdot q}{\|p\|\|q\|} = 1 - \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2}\sqrt{\sum_{i=1}^{n} q_i^2}}$$

Depending on the use case, these algorithms can achieve different results, while *Cosine Similarity* is the most common algorithm used in IR settings [BCB16]. In continued work on the Samos framework, Babur identifies that these originally included distance measures have different limitations [Bab18]. As mentioned, distance measures such as *Manhattan Distance* or *Euclidean Distance* are not normalized and make comparison of vector similarities (i.e., models in our case) hard. The previously used *Cosine Dissimilarity* provides such normalization and, in addition, is able to account for higher token frequencies that only exist because of different lengths in the input models as the *Cosine Dissimilarity* is length normalized. However, it does not properly reflect sizes of the analyzed models in the similarity, which is not desirable in certain cases. For example, when comparing a model containing two model parts A and B (e.g., two subsystems) with another model containing the same model parts twice, the *Cosine Dissimilarity* would calculate exactly the same value as their vectors would have the same angle, but with a different magnitude. However, this behavior is not desirable in model comparisons as the size of models has an impact on their similarity. Thus, we decided to use the *Bray-Curtis Dissimilarity* for our Corevid approach as it is able to properly reflect this criterion. The *Bray-Curtis Dissimilarity* for two vectors p and q of n dimensions is:

$$bray(p,q) = \frac{\sum_{i=1}^{n} |p_i - q_i|}{\sum_{i=1}^{n} (p_i + q_i)}$$

Executing the Clustering The VSM in combination with a selected distance measure enables application of clustering algorithms as the similarity returned for compared documents allows assignment to clusters of related documents. In literature these clustering algorithms are often separated into *flat* and *hierarchical* algorithms [MRS08, JD88]. *Flat clustering* algorithms, such as the *k-means* algorithm, have the disadvantage that the user has to know the number of existing clusters prior to the execution. Based on this information, the algorithm identifies a unique assignment of documents to these clusters. In contrast, *hierarchical clustering* algorithms identify no distinct categorization into clusters, but proximities of documents and create corresponding hierarchies.

The Samos framework applies unsupervised learning in form of *Hierachical Agglomerative Clustering (HAC)* to identify trees showing the relation of the documents. Different linkage strategies exist to combine clusters and create these trees. Examples are the *single linkage* and *complete linkage* strategies. Starting from one cluster per document, these approaches iteratively combine either the clusters of elements which contain a pair of elements with the *largest distance* (single linkage) or the shortest distance (complete linkage) [MRS08]. In contrast, the Samos framework uses the *average linkage* strategy as it builds a compromise between these extremes and, thus, realizes a rather conservative solution. This linkage strategy is described by the following formula, where *dist* is the selected distance measure (e.g., *Bray-Curtis Dissimilarity*).

$$averageLinkage(A,B) = \frac{1}{|A||B|} \sum_{a \in A} \sum_{b \in B} dist(a,b)$$

The resulting tree can be visualized in form of so-called *dendrograms*, which are subject to the interpretation by experts. In Figure 3.3, we show an exemplary dendrogram. The distance (i.e., the dissimilarity) of documents is shown by the horizontal lines connecting them and a connection at a lower value means that a higher similarity exists between the documents. In this case, document `1` and `5` can be seen as clear outliers compared to the remaining six documents as they have a large

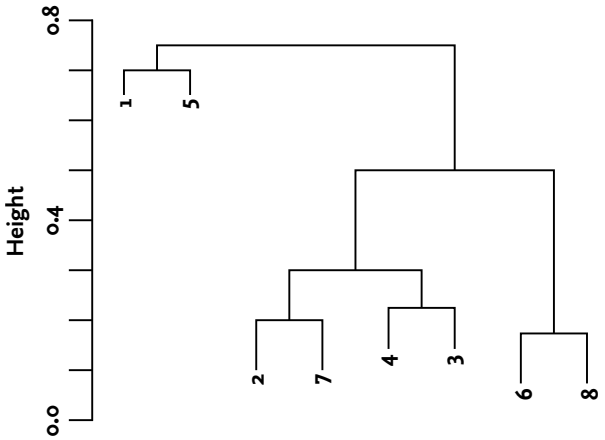


Figure 3.3.: Exemplary dendrogram.

distance to them. The number of clusters in this example depends on the interpretation of the experts. Documents `2`, `7`, `4` and `3` have a low distance to each other and, thus, can be seen as one cluster. However, documents `6` and `8` already have a higher distance to this cluster and it depends on the interpretation of the experts (e.g., based on the current use case) whether they represent another separate cluster or should be part of the first cluster.

In theory, it is possible to automatically define cuts in the corresponding dendrogram trees (e.g., at 30% distance) and consider elements below these thresholds as clusters. However, this is not desirable as the results are meant to guide developers during the exploration of the existing implementations and interesting information relevant to the experts might otherwise be lost. In fact, we argue that manual exploration and definition of clusters is desirable as users are forced to get an understanding of the existing implementations prior to applying techniques for low-level comparisons (e.g., on the state level in statecharts). This knowledge is helpful in subsequent steps of the analysis as developers are able to configure the algorithms for a detailed variability analysis (cf. Chapter 4) and a migration towards an SPL (cf. Chapter 5). Furthermore, the manual analysis not only allows to validate the results (e.g., that outliers are actually unrelated with other variants), but is also able to show interesting collaboration potential. For instance, developers might identify that the functionality of variants developed by other departments are closely related with their own variants and potential for collaboration exists.

3.4. Related Work

In literature, the larger part of existing approaches analyzes large sets of software variants on a low level of detail to extract variability models or migrate their implementation to an SPLs. Examples are the approaches for source code artifacts by Klatt et al. [KK12, KK13, KKK13, KKS14, KKW14, Kla14], Kästner et al. [KDO14, KDO11] and Fischer et al. [FLL+14, FLL+15, LFL+15, LLE17] as well as the approaches for models by Zhang et al. [Zha10, ZHM11, ZHM12, Zha14], Martínez et al. [Mar16, MZK+14], Rubin et al. [Rub14, RC12, RC13d] and Ryssel et al. [RPK10, RPK12a, Rys14]. In contrast to these approaches, our Corevid approach focuses on high-level relations to only provide an initial overview allowing cluster and outlier detection prior to a detailed analysis. For a detailed discussion of these approaches, we refer to Section 4.8 and Section 5.6 of this thesis.

Clustering Large Sets of Models To the best of our knowledge, only a small number of approaches exist that are directly related with our Corevid approach in the area of clustering model variants. Babur et al. [Bab16, BCB16, BCV+16, BC17, BCB18] use techniques from IR and statistical analysis to identify clusters of related models in sets of input models and meta-models. In continued work, the authors extend their techniques to identify clones in large meta-model and model repositories [Bab18]. An approach similar to the work by Babur et al. [Bab16, BCB16, BCV+16, BC17, BCB18] is the work by Basciani et al. [BDD+16]. This approach focuses on model repository management and executes automated categorization of contained models using HAC to visualize their relations. Our Corevid approach builds upon the Samos framework by Babur et al. [Bab16, BCB16, BCV+16, BC17, BCB18] to detect clusters and outliers and, thus, is a direct application of their techniques to a concrete problem. Here, the Corevid approach can exploit the framework's capabilities to analyze the relations between model variants and support developers during the detection of clusters and outliers in the input models. As a result, it is capable of providing information to select relevant variants for a detailed analysis of low-level relations.

Another approach exists in the area of business process models, where Dijkman et al. [DDD+11] use clustering on footprints of business processes to search for such models in large repositories. While the underlying techniques are quite similar, the proposed approach is only focused on a single notation and, thus, not directly applicable for our purposes. In contrast, the Samos framework used for the Corevid approach provides an interface for custom extraction schemes. This allows adaptation for new languages and easy reuse of the framework's generic clustering facilities.

Another approach by Bislimovska et al. [BBB+14] uses typed n-grams to calculate similarities during the comparison of model nodes in the context of indexing and searching of web-based model repositories. While the approach relies on typed n-grams, it focuses on searching a model repository based on user-specified queries. In contrast, we are interested in identifying clusters of related models and outliers to guide users during the exploration of the existing model variants.

Clustering during Low-Level Variability Identification Furthermore, different approaches exist that apply clustering during the identification of low-level variability relations between models. Ryssel [Rys14] clusters Matlab/Simulink subsystems from model variants to find the correct relation between them prior to identifying their low-level variability. Similar to our Corevid approach, the authors argue that manual interaction to determine the correct relations is inevitable in many cases as finding an optimal solution using automatic cutting of dendrograms can result in undesired results. In addition, Strüber et al. [SRA+16] use clone detection techniques combined with clustering to identify model transformation rules that can be combined into merged rules comprising variability. To this end, the authors use automatic cutting with a user-specified threshold to automatically merge corresponding variable transformation rules without additional user interaction. While the author's approach is able to cluster and merge variants of transformation rules, our Corevid approach is focused on providing guidance in a reverse-engineering scenario to explore the analyzed variants. Thus, the completely automatic approach by Strüber et al. [SRA+16] is not applicable as we are explicitly interested in the manual interaction with the identified relations. Furthermore, Reinhartz-Berger et al. [RZW16] use clustering to group similar variation points and suggest corresponding realization mechanisms using an ontology-based approach. In contrast to our Corevid approach, these authors are not interested in showing relations between complete variants, but only for parts of the analyzed implementations. Thus, the approach is not applicable in

our case as we focus on suggesting relations between complete model variants. Another approach by Alalfi et al. [ACD14] uses clustering to identify patterns of cloned parts in Matlab/Simulink models and to visualize them. While this approach is able to identify similar subparts of such models, it is limited to a single notation (i.e., Matlab/Simulink models) and does not consider complete model variants. In contrast, our Corevid approach is easily extensible through the interface for extension schemes of the Samos framework and presents relations for complete model variants to the user.

3.5. Chapter Summary

Previously, analyzing large sets of models with unknown high level relations between them was a tedious manual task. Developers had to manually inspect all existing implementations and identify which variants are related prior to applying a detailed analysis of low-level variability relations (e.g., which transitions in statecharts are alternative). Using the described Corevid approach it is now possible to categorize large sets of existing variants into clusters and outliers. Based on the generated dendrograms, developers are able to explore the relations between variants and to remove outliers. Furthermore, it is possible to select clusters of related variants and to only focus on subsequent detailed variability analysis of the related variants in these subsets. This way developers are able to create viewpoints to specifically focus on these variants and leave irrelevant implementations (e.g., containing completely unrelated features) aside for their current task (e.g., fixing a bug in a specific feature).

4 Executing Custom-Tailored Variability Mining for Different Block-Based Languages

The contents of this chapter are largely based on the work published in [WBC+18, WWP+16, WSS16, SWS+17].

Summary *Analyzing low-level relations between large sets of model variants (e.g., in a clone-and-own scenario) involves large effort as common and differing parts have to be manually identified. For instance, identifying all variants containing a bug, which was identified for one of the variants, involves time-consuming manual comparisons. Depending on the considered models such an analysis might even be doomed to fail as industrial models often contain hundreds or thousands of model elements and too many relations might have to be analyzed. To overcome these problems, we describe our variability mining approach in this chapter, which is capable of (semi-)automatically identifying such relations. By providing guidelines with corresponding tooling and a compare process relying on user-adjustable metrics, we allow easy adaptation of the described algorithms for new block-based languages and differing settings (e.g., models of a different domain).*

Deriving variants from an SPL involves additions, deletions and modifications of low-level implementation artifacts (i.e., the used model elements) to create the desired behavior. In case of statecharts these changes could, for example, modify names of existing states, remove unneeded transitions and add further regions realizing complete implementation features. Thus, for a successful migration of a family of related variants (e.g., created using clone-and-own approaches) to managed reuse in an SPL it is essential to identify which model elements of the analyzed implementations are shared and which are differing. We refer to variability relations on this low level of abstraction as fine-grained variability relations.

Definition 4.1: Fine-Grained Variability Relations

Fine-grained variability relations describe the variability between elements by means of the low-level implementation artifacts of the analyzed language (e.g., states or transitions of statecharts). Fine-grained variability information gives details on the *mandatory parts* (i.e., common to all variants) and most importantly the varying parts that differ across the family's variants. Consisting of *alternative parts* (i.e., mutually exclusive across variants) and *optional parts* (i.e., only present in certain variants), these varying implementation parts represent the low-level configuration options for model variants derived from the created SPL.

In Figure 4.1, we give an overview of our (semi-)automatic variability mining approach to identify and analyze such fine-grained variability relations between related model variants of large families.

As our approach focuses on identifying such variability information for complete families of related variants, we refer to it as the Family Mining approach in the remainder of this thesis. Our Family Mining approach expects a set of variants as input and analyzes them in a pairwise manner to iteratively create a 150% model (cf. Definition 2.10) storing the fine-grained variability relations of all compared variants. For two analyzed variants the approach executes the following three phases to create a 150% model that afterwards either is used as input for the comparison with the next variant or returned for an analysis by a domain expert or the migration to an SPL:

- *Compare Phase:* This phase traverses the currently analyzed variants and identifies possible variability relations by comparing specific model parts with each other. By using a metric specifically created for the analyzed modeling language and tailored towards the domain experts expectations, a similarity is calculated for each two compared model elements.
- *Match Phase:* The previously identified variability relations might be ambiguous (e.g., when multiple elements from one model are possibly related to a single element from another second model). Thus, this phase identifies for each ambiguity a distinct relation between exactly two model elements from the analyzed models.
- *Merge Phase:* Based on these distinct relations, this phase merges a 150% model storing all model elements from the compared model variants together with annotations showing their explicit variability relations and their containing variants.

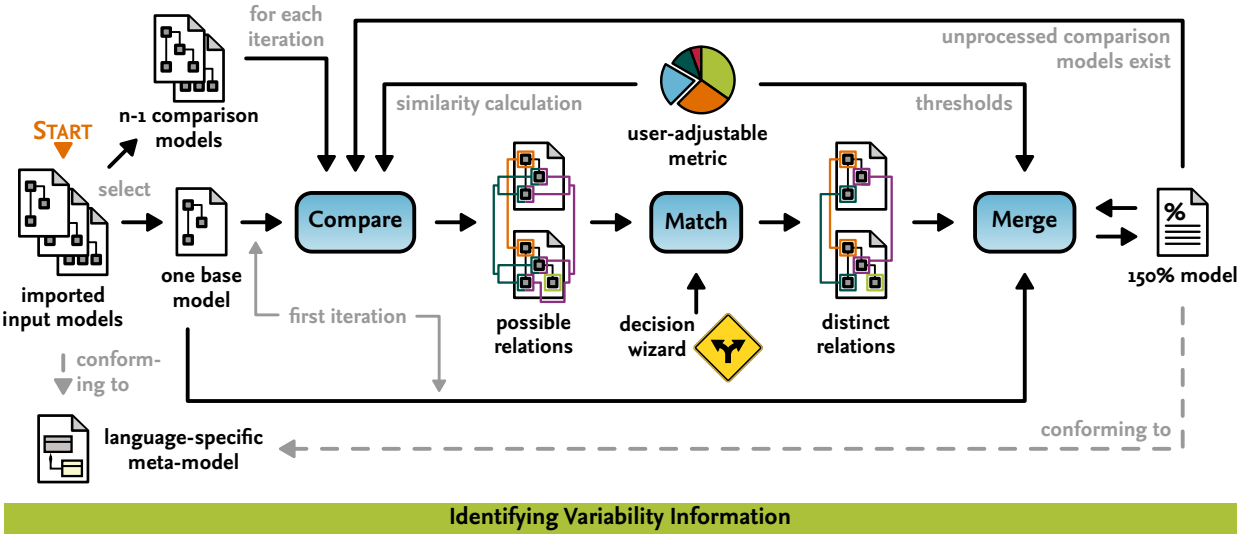


Figure 4.1.: Workflow of the Family Mining approach to identify fine-grained variability information.

Chapter Outline The focus of our Family Mining approach lies on the extensibility of the algorithms for new languages (e.g., in other domains) and their adaptability to other variability mining scenarios (e.g., where additional information on the compared variants is available). Thus, our Family Mining approach allows domain experts to adapt the existing Family Mining implementation to changed scenarios by influencing each of the three phases. Domain experts can influence the comparisons of model elements by adjusting the metric that is used to calculate their similarity during the *Compare Phase*. During the *Match Phase* the matching process can be influenced by a

customized decision wizard in situations where no distinct relation can be identified for ambiguous elements (i.e., they are regarded as equally similar according to the metric). Here, the decision wizard either allows to provide a resolution algorithm to automatically resolve the ambiguity or to manually select the desired resolution. In addition, the metric's thresholds used to derive the explicit variability relations between elements during the *Merge Phase* can be adjusted to generate 150% models conforming to the expectations of domain experts.

Our current implementation of the Family Mining supports variability mining for different block-based languages commonly used in industry, such as, The MathWorks Matlab/Simulink models, *Function Block Diagrams (FBDs)* as part of the IEC 61131-3 standard [Int13] and statecharts in tool-specific notations. These include the statechart notation by Harel [Har87], The MathWorks Stateflow, ETAS ASCET[1], IBM Rational Rhapsody[2], Esterel Technologies SCADE Suite[3], Yakindu[4] and the UML standard as developed by the OMG [Obj15]. In addition, we implemented the algorithms in a generic way to allow for easy adaptation of our Family Mining approach. Furthermore, we created guidelines consisting of four steps to describe how our algorithms can be made available for additional languages. An overview of these steps can be found in Figure 4.2.

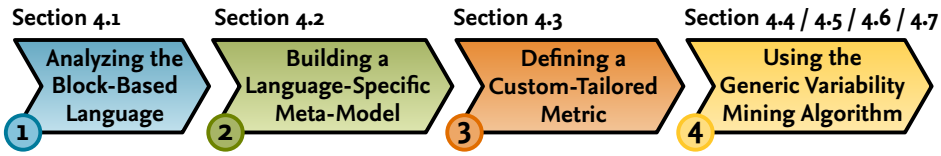


Figure 4.2.: Four steps of the guidelines to adapt the Family Mining algorithm for new languages.

In the following sections, we explain the details of each step by applying them to our running example (cf. Section 2.2) and discuss how large parts of the needed infrastructure can be generated:

- *Section 4.1:* Starting with a detailed analysis of the new block-based language that should be added to the Family Mining approach, we first analyze which elements have to be considered during the mining to identify sensible variability relations between compared variants.
- *Section 4.2:* Based on the insights gained about these relevant model elements, we build a meta-model representation allowing to abstract from information unnecessary for the similarity comparison (e.g., the color or position of model elements).
- *Section 4.3:* To allow comparisons between elements of the created meta-model, we define a custom-tailored and easy adjustable metric which calculates corresponding similarity values.
- *Section 4.4:* To execute the variability mining, we realized a generic Family Mining algorithm for the comparison of model variants (cf. Figure 4.1).
- *Section 4.5:* To identify distinct variability relations between the compared model variants, we have to execute our generic match algorithm (cf. Figure 4.1). Both, the comparison and the matching algorithms were realized in a generic way to reduce effort during adaptation of our Family Mining algorithms for new languages.

[1] https://www.etas.com/en/products/ascet-developer.php
[2] http://www.ibm.com/software/awdtools/rhapsody/
[3] http://www.esterel-technologies.com/products/scade-suite/
[4] https://www.itemis.com/en/yakindu/state-machine/

- *Section 4.6:* To merge the identified variability information in a 150% model, developers have to implement corresponding steps (cf. Figure 4.1). We give details on how to realize such algorithms for new languages during the adaptation of our Family Mining.
- *Section 4.7:* While our Family Mining algorithms already identify all variability relations correctly for large sets of input models, we identified during work with industry partners that certain more complex and less frequently occurring changes to models in clone-and-own scenarios might not be detected. To improve the variability information identified by our Family Mining approach even more, we discuss additional algorithms that identify hierarchy shifts of model elements and insertions causing horizontal dispersions between model elements.
- *Section 4.8:* This section provides a comparison with related work in the area of mining fine-grained variability relations for large model families.

4.1. Analyzing the Block-Based Language

As mentioned in Section 2.1 MDSD techniques are used to abstract from complex data and to allow developers to effectively analyze data without considering details irrelevant for their current task [SV06]. For this reason, we decided to realize our Family Mining approach based on MDSD techniques by importing the input model variants to instances of a meta-model specifically created for the used block-based language. This meta-model allows us to abstract from information that is irrelevant for the analysis of the variants' variability relations and, thus, reduces the overall analysis effort. Such irrelevant details could be the colors used for visualizing the model elements or the x and y coordinates used to layout them. Furthermore, using a model-based solution enables us to realize generic Family Mining algorithms that can be easily adapted for new languages by exchanging the used meta-model.

To find such an abstraction from block-based languages used to model the input variants, we first execute a detailed analysis of all concepts used by the corresponding language. We define such language concepts as follows.

Definition 4.2: Language Concepts

Model-based languages consist of different *language concepts* that allow modeling complex systems and their behavior. These concepts consist of *language elements* with corresponding *element properties* that allow to influence the elements' behavior or appearance in corresponding visualizations.

Examples for language elements are *states* or *transitions* that, for example, allow to model the system states of a statechart and transitions between them. Corresponding examples for element properties are the *color* of states or the *actions* executed for transitions. During the analysis of the language concepts, it is important to keep in mind that the Family Mining algorithm can only consider data that is modeled in the meta-model representation of the language. Thus, finding the right level of abstraction is a crucial task, which decides on the meaningfulness of the variability relations identified by our Family Mining approach. On the one hand, providing too little information might reduce the validity of the results as key language aspects are not considered and

incorrect variability relations might be identified. On the other hand, considering too much information during the comparisons might result in unnecessary effort to provide this data and can also negatively affect the runtimes of the Family Mining algorithms. Thus, we consider the language analysis as a completely manual task, because profound knowledge about the language's concepts is needed to select the right amount of details.

4.1.1. Searching for Existing Meta-Models

Before starting a detailed analysis of potentially large language documentations and other resources, we recommend considering existing meta-models from other sources. Searching for existing meta-models can reduce the overall effort to adapt Family Mining for a new language and, thus, might save costs as this theoretically allows to skip to *Step 3* of the guidelines (cf. Section 4.3) in Figure 4.2. Nevertheless, we strongly encourage developers, even in cases where a meta-model exists, to use the ideas in subsequent sections as a chance to identify the concepts' relevance for the similarity of compared model elements. This way, the effort of defining a metric enabling the model comparisons in *Step 3* of the guidelines (cf. Section 4.3) is eased. We identified three possible sources that should be considered when checking for opportunities to reuse existing meta-models:

1. In case of adapting Family Mining for variability mining in a company it can be worthwhile to check the company's repositories for projects that already use matching language meta-models. Reusing such company property also reduces the chance of conflicting licenses and increases the chance of having domain experts available that can help during later steps of the adaptation (e.g., the metric definition in *Step 3* in Section 4.3).

2. In case of adapting Family Mining for a language that already relies on a meta-model it should be possible to easily adapt the existing meta-model to the requirements of our Family Mining approach (cf. Section 4.2.2 and Section 4.2.3). For example, Yakindu statecharts rely on an EMF-based EMOF meta-model that is provided as an open source project under the *Eclipse Public License* on GitHub[5].

3. Another possibility are meta-models from publicly available sources, such as, the AtlanMod Meta-Model Zoo[6], which collects existing meta-models under an open source license. Meta-models from such external sources should be checked for their completeness and quality to ensure their usability for Family Mining. In addition, these meta-models also have to be adjusted to the requirements of our Family Mining approach (cf. Section 4.2.2 and Section 4.2.3).

4.1.2. Analyzing Relevant Language Concepts

In case no existing meta-model could be found, a structured analysis of the language's concepts has to be executed to a) get an overview and deep understanding of all used language concepts and b) decide which of them are relevant for a detailed variability analysis of model variants. Overall, the language analysis enables developers to collect all necessary information for building a meta-model in *Step 2* of our guidelines (cf. Section 4.2). Even when the developers adapting Family Mining for the new language are well familiar with developing corresponding model implementations, we recommend to consider the following analysis possibilities. Otherwise, less frequently used language

[5] https://github.com/Yakindu/statecharts
[6] http://web.emn.fr/x-info/atlanmod/index.php?title=Zoos

concepts might be ignored and the Family Mining results at later stages might not consider all relevant details leading to inadequate results. In addition, defining the correct scope for the created meta-model not only influences the quality of the generated results, but effectively allows to reduce executed comparisons to language concepts actually used in the considered domain or company. For example, Matlab/Simulink allows to use so-called *Function Blocks* to define the block's functionality by using source code in a *General Purpose Language (GPL)* – e.g., C or C++. In case such blocks are not used in the considered setting, the scope can be reduced and no complex algorithm has to be implemented to compare corresponding source code artifacts.

For the execution of structured language analyses, we identified a number of possible sources:

Language Specifications Formal language specifications provide a highly reliable source of information as they normally give a detailed (or even complete) description of the language and are issued by the developing company or standardization committee. Due to their high degree of formalization, they leave little scope for incorrect interpretation. Examples are standards, such as the IEC 61131-3 standard [Int13] for *Function Block Diagrams (FBDs)* or the UML standard [Obj15]. However, such standards only show the capabilities of the language and often comprise a large number of concepts that are less frequently used in practice. Thus, we recommend to involve domain knowledge to define the right scope for a company-specific or domain-specific adaptation of our Family Mining algorithms (e.g., by ignoring unused concepts).

Language Guides Language guides issued by the developing company or standardization committee of the language provide another reliable resource. Most importantly they give concrete examples how the different language concepts should be used in practice and the lower level of formalization might help to get an initial understanding of the language. On the other hand, the level of detail and relevance for a company-specific or domain-specific adaptation of our Family Mining algorithms depends on the corresponding guides and other sources should be involved.

Existing Implementations Existing implementations from the language's domain provide a high level of detail how the different concepts are used in practice. Thus, such implementations have a higher relevance for company-specific or domain-specific adaptations of our Family Mining algorithms. However, the high level of detail due to the possibly large real-world implementations can be a hurdle and a sensible selection of examples is advised.

Development Conventions Company-specific or domain-specific development conventions have a high relevance for finding the right scope during the Family Mining adaptation of new languages as realistic best practice scenarios are shown. However, due to the exemplary character of such conventions, the information content largely depends on the actual number of introduced conventions.

Interviews of Experts (Semi-)structured interviews [Pat87] of domain experts provide the chance to identify valuable information for finding the right scope during the Family Mining adaptation in company-specific or domain-specific settings. Defining a set of relevant questions regarding different setting-specific topics (e.g., which language concepts have a high impact on the similarity of compared elements or which conventions are most relevant) prior to questioning the domain experts allows to elaborate questions to gain insights in the domain. Depending on the chosen questions different degrees of information and formalization can be achieved. *Closed questions* give the chance to confirm details that were previously gained from other resources or to let experts rank

Source	Quality	Relevance for Specific Adaptation	Level of Detail	Degree of Formalization
Language Specifications	++	○	++	++
Language Guides	+	○	○	+
Existing Implementations	+	+	++	○
Development Conventions	+	++	○	+
Interviews of Experts	+ / ○	++	+	+ / ○
Discussions with Experts	○ / –	++	○	– –
Tutorials & Guides	○ / –	○	○	– –

+ / ++ *high / very high assessment* ○ *medium assessment* – / – – *low / very low assessment*

Table 4.1.: Possible sources for a structured language analysis together with an assessment of their quality.

concepts according to their relevance for the variability mining. *Open questions*, on the other hand, allow to get detailed but less formal answers (e.g., about the usage of certain language concepts or conventions). Overall, the quality of gained knowledge highly depends on the participants of the interviews (e.g., their experience) and the asked questions.

Discussions with Experts Discussions with domain experts provide a less formal source for details about languages. However, they give developers adapting Family Mining for a new language the chance to clarify details about used development conventions in an informal way. Overall, the quality of gained knowledge mostly depends on the participants (e.g., their experience).

Tutorials and Guides Tutorials and guides from third party sources are an additional resource to understand the details of language concepts and how they are used. However, the quality of such tutorials and guides highly depends on the sources. While generally books from experts are more reliable, tutorials from the Internet often lack the formal details and, thus, might be less helpful. In addition, the relevance of shown examples for a company-specific or domain-specific adaptation of our Family Mining algorithms highly depends on the focus of the corresponding source.

In Table 4.1, we summarize all identified and discussed sources together with an assessment regarding their information quality, relevance for a company-specific Family Mining adaptation, level of detail and degree of formalization.

Depending on the available sources and their quality it can be sensible to combine different sources on the analyzed language to get a broad overview. For example, for a language with a formal specification it can be sensible to combine the corresponding information with insights from the domain (e.g., through interviews of experts or discussions with them) to scope the relevant language concepts to the ones that are actually used. In another setting it could be sensible to analyze existing implementations using the language with knowledge from domain experts (e.g., through interviews of experts or discussions with them). Furthermore, it can be sensible to check whether other similar language dialects are used in the domain or company. By aggregating the concepts of the different dialects it might be possible to create a single meta-model supporting multiple dialects. For example, we were able to combine information on the statechart notation by Harel [Har87],

The MathWorks Stateflow, ETAS ASCET, IBM Rational Rhapsody, Esterel Technologies SCADE Suite, Yakindu and the UML standard as developed by the OMG [Obj15] to create a meta-model supporting Family Mining of these seven statechart dialects [WSS16]. In any case, we recommend to continue with the selection of relevant language elements and properties only after a broad understanding of the language concepts and their intentions exists.

4.1.3. Selecting Relevant Language Elements and Properties

After identifying all language concepts with corresponding details the gained insights have to be used to find a classification of the language's elements and their properties according to their relevance for identifying variability relations between variants of such models. The classified concepts afterwards are used to create a matching meta-model in *Step 2* (cf. Section 4.2) and to define a metric in *Step 3* (cf. Section 4.3) of our guidelines. Language concepts can be classified as *relevant* and *irrelevant* concepts.

In accordance to Definition 4.2 *relevant language concepts* are defined by language elements and properties that necessarily have to be considered to identify sensible variability relations between compared model variants during the Family Mining. Examples are language elements that define the functionality of models (e.g., *states* or *transitions* in statecharts) and properties that influence their behavior (e.g., *events* of transitions) or make them distinguishable (e.g., *names* of states). In contrast, *irrelevant language concepts* are language elements or properties that do not provide additional information about compared model variants and, thus, do not have to be considered to identify sensible variability relations during Family Mining. Examples are language elements that do not define any functionality in models (e.g., *Annotation Blocks* used in Matlab/Simulink to add comments) or properties that do not influence the elements' behavior (e.g., the *color* of states used in statechart development tools). In addition, certain language elements or properties might be regarded as irrelevant, because they are not used in a certain setting (e.g., a specific domain or company) and, thus, do not need to be considered, although, they might be relevant in other settings. Regarding such elements as irrelevant can significantly reduce the effort to adapt Family Mining for a new language and positively influences runtimes as less elements might need to be processed.

At this point, we highlight the special role of element *names*. We normally would regard them as irrelevant with respect to the models behavior, because they generally have no impact on the model's execution. However, they allow us to make elements distinguishable that would otherwise appear to be the same (e.g., two states without distinguishable properties).

Furthermore, we emphasize the relevance of element *neighborhoods* during model comparisons. Generally, the neighborhood of an element is defined by the incoming and outgoing connections surrounding it as well as the corresponding linked elements. Considering these neighborhoods during model comparisons, it is possible to "localize" an element in a model and check whether its surroundings are similar to the compared element in another model. Thus, this additional information further helps to make model elements more distinguishable that otherwise would appear to be the same. Depending on the use case or language, the accuracy for comparing the location of two elements might be improved by not only considering the direct neighbors of elements, but also analyzing the neighbors of these direct neighbors. In theory, it is possible to consider even wider circles around the compared model elements. However, it is important to keep in mind that each additionally considered level of surroundings will increase the complexity for comparisons, because

depending on the size of the neighborhoods the processed data might increase exponentially. As a result, considering the direct neighbors should be sufficient in most cases.

In addition to relevant and irrelevant language concepts, there might also exist elements that can be transformed to other representations. These elements are regarded as *syntactic sugar*[7], if they a) have to be classified as *relevant* and b) can be transformed back and forth between their original representation and a representation with equivalent behavior without changing the model's overall behavior. An example for such syntactic sugar language concepts are *ModelReference Blocks* in Matlab/Simulink models. These blocks allow to include Matlab/Simulink models that were stored in external files. These blocks can easily be transformed to an equivalent representation, where a *Subsystem Block* containing the contents from the external file is inserted instead of the original *ModelReference Block*. For these syntactic sugar language concepts, the developer has to find a trade-off between cluttering the created meta-model with elements that could be transformed to equivalent representations and implementing such transformations. On the one hand, adding further elements to the meta-model results in a higher complexity during comparisons as, for example, additional algorithms would be necessary to compare a *ModelReference Block* with normal blocks. On the other hand, transforming back and forth between different representations adds additional complexity to the import and export of models as the Family Mining results should be presented in a similar notation as the input models to not confuse developers.

When executing the language analysis for multiple dialects of the same language (e.g., differing statechart notations) to build a unifying meta-model, it is crucial to identify equivalent concepts with differing names. Otherwise, a high chance exists that redundant concepts are created under different names. For example, the UML as developed by the OMG [Obj15] standard uses *initial pseudostates*, while ETAS ASCET calls them *start states*. Without aligning these concepts, developers might end up with a solution that contains both concepts, although they are both used to indicate the execution start of statecharts. On the other hand, it is important to execute a thorough analysis of all notation concepts to make sure that the created meta-model is expressive enough to create models for the considered languages.

In Table 4.2, we show an excerpt from the exemplary reasoning about the seven dialects analyzed for our statechart meta-model (i.e., the notation by Harel [Har87], The MathWorks Stateflow, ETAS ASCET, IBM Rational Rhapsody, Esterel Technologies SCADE Suite, Yakindu and the UML standard as developed by the OMG [Obj15]). The table shows for each language element the corresponding properties together with a reasoning that led to the assessment. For the sake of clarity, we only show an abbreviated version of the concepts used by these dialects. This allows us to discuss the general concepts of our Family Mining algorithms without impairing the understandability by showing details unnecessary to explain them.

Given that the language analysis was executed exhaustively and a sensible categorization in relevant and irrelevant language concepts was found, it is possible to create an appropriate meta-model for Family Mining in *Step 2* of our guidelines.

[7] According to a travel report by Dijkstra [Dij78] and Abelson et al. [AS96] the term "syntactic sugar" was first coined by Landin [Lan64] in 1964.

Language Element	Assessment	Reasoning
States	✓	**model the states of systems**
• *name*	✓	*makes states distinguishable*
• *hierarchical / parallel*	✓	*whether one or multiple regions exist*
• *initial state*	✓	*distinctly shows the execution start on each hierarchy level*
• *final state*	✓	*allows to terminate the current execution*
• *interface*	✓	*the incoming and outgoing transitions are a state's interface*
• *color*	✗	*does not add behavior / only for visualization*
• *x / y coordinates*	✗	*do not add behavior / only for visualization*
Regions	✓	**enable modeling of hierarchy**
• *name*	✓	*makes regions distinguishable*
• *sub states*	✓	*→ cf. States*
• *sub transitions*	✓	*→ cf. Transitions*
Transitions	✓	**allow transitioning between different system states & model the executed behavior of systems**
• *label*	✓	*allows to define the transition's behavior*
• *multiple labels*	Ⓢ	*allow to add multiple labels concatenated with logical `OR`s → transform to multiple transitions with single labels*
• *x / y coordinates of bend points*	✗	*do not add behavior / only for visualization*
Transition Labels	✓	**allow to assign conditional functionality to transitions**
• *events*	✓	*trigger the execution of the transition*
• *conditions*	✓	*have to be fulfilled prior to executing the actions*
• *actions*	✓	*executed after the conditions were fulfilled*

✗ *irrelevant language concept* Ⓢ *syntactic sugar* ✓ *relevant language concept*

Table 4.2.: Exemplary reasoning about the relevance of statechart concepts for Family Mining.

Language Concept	Meta-Model Concept	Example
Language Element	`EClass`	*states* in *statecharts*
Element Property	`EAttribute`	*names* of *states* in *statecharts*
Relations	`EReference`	*transitions* have *source* and *target states* in *statecharts*
Specialization	*Inheritance*	*initial states* are a specialization of normal *states* in *statecharts*

Table 4.3.: General guidelines to represent identified language concepts as meta-model concepts.

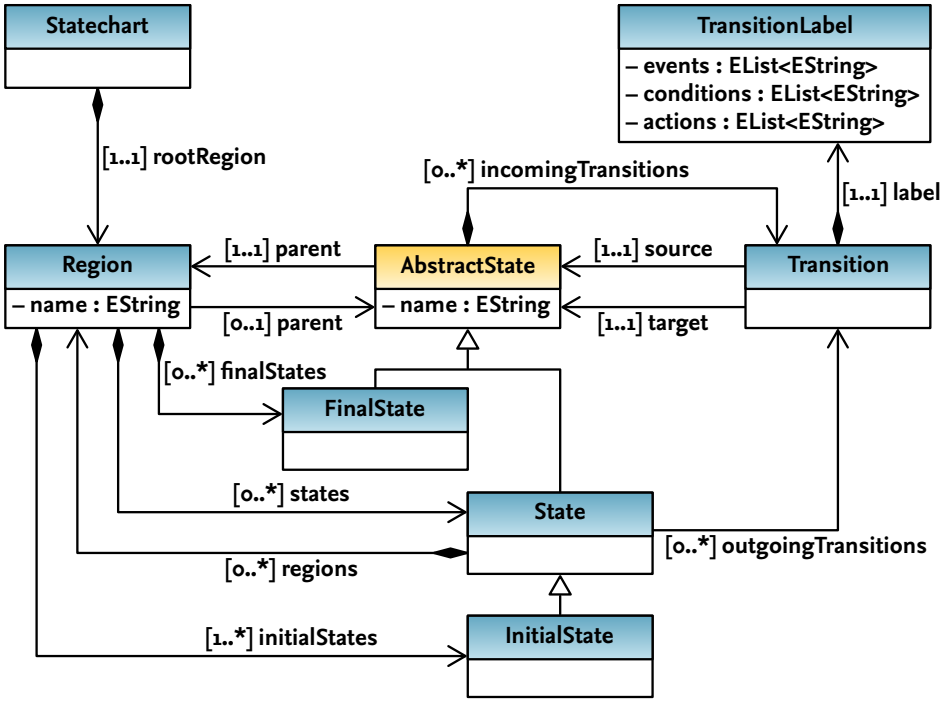


Figure 4.3.: Simplified meta-model used for Family Mining of statechart variants. We previously showed this meta-model in Figure 3.2 to explain the Corevid approach, but show it again for convenience.

4.2. Building a Language-Specific Meta-Model

Based on the detailed language analysis in *Step 1* of our guidelines (cf. Section 4.1) it is possible to build a corresponding meta-model or modify an existing meta-model to the requirements of the Family Mining. Before starting the meta-model creation, it is crucial to select a suitable meta-modeling language. For the meta-models created in our projects, we selected EMF with the meta-modeling notation Ecore as both provide seamless integration with Eclipse and the provided tools (cf. Section 2.1). Thus, the following details will be given with respect to the Ecore meta-model.

4.2.1. Building the First Meta-Model Version Based on the Analysis Results

Using the selected meta-modeling language, the meta-model should be created based on a reasoning, such as our detailed analysis in Table 4.2. Generally speaking, the guidelines in Table 4.3 can be applied to identify the correct representation of language concepts in a meta-model. Each relevant concept should be represented by a meta-model class (i.e., an `EClass` in Ecore) with attributes modeling the concept's relevant properties (i.e., `EAttributes` in Ecore). However, in certain cases, it makes sense to deviate from these rules and, thus, we regard them only as general guidelines.

In Figure 4.3, we show a simplified version of our meta-model used for the Family Mining of statecharts. For the sake of clarity, we omitted details that are not necessary to understand our guidelines or the Family Mining algorithms described in this thesis. In the appendix, we show our complete meta-model representations for Family Mining of:

- All seven statechart notations supported by our statechart Family Mining (cf. Figure C.1).
- Matlab/Simulink models and IEC 61131-3 *Function Block Diagrams (FBDs)* [Int13] (cf. Figure D.1).

For the simplified statechart meta-model, we identified in Table 4.2 that properties can distinguish between *states*, *initial states* and *final states*. Here, we decided to deviate from our guidelines. Instead of using `boolean` attributes in a single class, we model all states as classes by using an `AbstractState` class as generalization for the three classes. This allows easier representation of relations between them. For example, *states* can contain *sub regions*, which is also the case for *initial states* but not for *final states*. Thus, the `InitialState` inherits from the `State`, which in turn inherits from the `AbstractState`, while the `FinalState` directly inherits from `AbstractState`. This allows an efficient modeling of *transitions*, as all *states* can have incoming transitions, but only *states* and *initial states* allow outgoing transitions, because *final states* end the current execution. Such a behavior can be directly modeled using the described inheritance between states. In contrast, we decided for our Matlab/Simulink meta-model that *Subsystem Blocks* are not represented by an additional `EClass`, but modeled by using `Block` class instances with the *type* attribute *Subsystem*.

In case syntactic sugar was identified during the initial language analysis, developers have to clearly define transformations between the input notation and the created meta-model to document how corresponding model elements can be transformed back and forth during import and export. For example, a transition having multiple labels that are concatenated with logicals `OR`s can be transformed to multiple transitions with each containing exactly one of the labels (cf. Table 4.2). To enable transformation back to a representation with a single concatenated label, an annotation has to be added during import to store for each of the single label transitions that they originally belonged to a single transition with multiple labels. A suitable solution could be creating a unique identifier (e.g., a *Universally Unique Identifier (UUID)*) per transition with multiple labels and adding it to each of the created single label transitions. This way a clear relationship between transitions is obtained. Without such countermeasures, information is lost during import and developers might get confused when the exported models are not transformed back to match the original input files.

4.2.2. Adding Meta-Model Mechanisms to Store Variability Information

When building the meta-model representation for an analyzed language, it is important to keep specific requirements of 150% models in mind to make the created meta-model *variability-aware* and, thus, usable for Family Mining.

Definition 4.3: Variability-Aware Meta-Models

A meta-model is *variability-aware* if it is capable of storing semantically correct models from the original modeling notation as well as corresponding 150% models. To allow creation of such 150% models during Family Mining without loosing identified variability information, the corresponding meta-model needs mechanisms to:

1. *Store variable model elements* in the 150% model.
2. *Annotate the variability* of model elements as *mandatory*, *alternative* or *optional* elements.
3. Annotate that a number of model elements form a *group of alternative elements*.
4. Annotate the *containment of model elements* in specific model variants.
5. Annotate *differing values of element attributes* across the model variants.

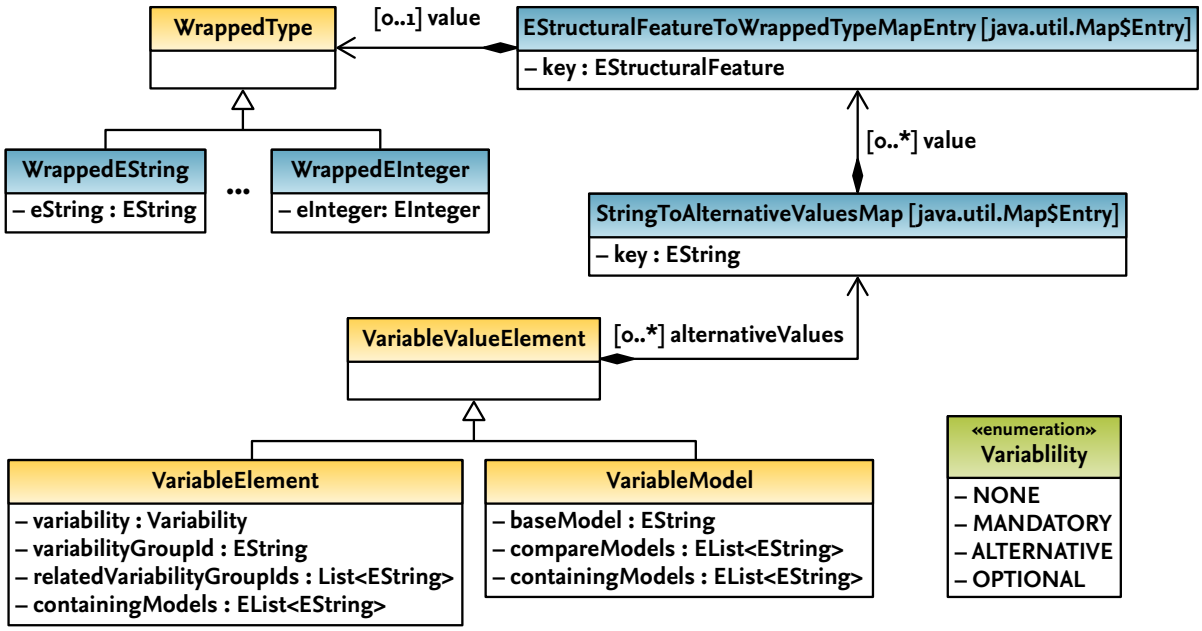


Figure 4.4.: Excerpt from our meta-model enabling variability-aware meta-models following Definition 4.3.

To ease creation of variability-aware meta-models we provide a *base meta-model* following Definition 4.3, which provides basic mechanisms to model variability and can be used by inheriting from its classes. In Figure 4.4, we show an excerpt of the base meta-model, which we use for Family Mining of statecharts as well as Matlab/Simulink models and IEC-61131-3 FBDs [Int13]. We provide classes for `VariableModels` storing information about the comparisons and the names of all *contained models* (i.e., all models stored in this model instance). The `VariableElement` class allows to annotate variability information for model elements by providing mechanisms to store the *variability* of elements, the *id* of their *alternative variability group* and the models *containing* the corresponding element. In addition, the `VariableValueElement` class allows to store modified attributes for inheriting model elements by mapping model names to lists of modified attributes (e.g., when the name of a state changed). To this end, the `StringToAlternativeValues` map allows to store for each `String` (i.e., a model variant) a list of `EStructuralFeatures` (i.e., the element's attributes) containing the alternative values of the different variants (i.e., by storing them as a `WrappedType`). While this base meta-model allows modeling of variability, our approach is not bound to this solution and we allow for custom solutions as long as they follow the points in Definition 4.3. For details on the complete base meta-model, we refer to Figure A.1.

The provided base meta-model addresses points 2 – 5 from Definition 4.3 and, thus, already solves all points regarding the annotation of identified variability information. However, point one from the definition (i.e., support to store variable elements) can only be implemented when creating the references between language elements in the meta-model. For example, when adding the `initialStates` reference between the `Region` class and the `InitialState` a meta-model without variability-aware extensions would only allow to store a *single* initial state per region as classic statecharts require a distinct execution start. However, in case of variability-aware meta-models this strong requirement has to be softened, because in a variability mining scenario it could be possible that alternative initial states are identified. Thus, the meta-model has to be capable of stor-

ing *multiple* initial states. Although this is a clear violation of the classic statechart definition, it is necessary to enable the correct creation of 150% models according to Definition 2.10. Nevertheless, it is important to keep the resulting implications in mind, as changing the allowed cardinality from $[1..1]$ to $[1..*]$ might have implications on other artifacts used during the development of models. For instance, when using OCL constraints to validate specific characteristics of model instances derived from the created meta-model, it is important to keep in mind that such changes to the meta-model can make other assumptions void. A concrete scenario could be the validation of contained initial states in model instances of the statecharts meta-model. Developers might have assumed that only a single initial state can exist per region and created the OCL constraint in Listing 4.1. When applying this constraint to a 150% model instance of the statechart meta-model, it is possible that a violation is identified as multiple alternative initial states might exist in regions of a 150% model. Thus, we recommend keeping a clear separation between meta-model instances storing model variants and instances storing 150% models to eliminate such possibly unexpected results and unnecessary debugging.

```
1 context Region
2 inv: self.initialStates->size() = 1
```

Listing 4.1: Exemplary OCL constraint checking the number of initial states per statechart region.

When using an existing meta-model for the adaptation of Family Mining for a new language, all points from Definition 4.3 have to be addressed to make the reused meta-model variability-aware. This can be done by inheriting from our base meta-model and adjusting the cardinalities of references in the existing meta-model, if necessary. However, in situations where the meta-model cannot be changed (e.g., when company regulations prohibit it or changing the meta-model might break other processes) different solutions have to be applied. A possible approach to circumvent such restrictions is adapting a copy of the meta-model and using model-to-model transformations (cf. Section 2.1) between the two meta-models.

4.2.3. Lifting the Created Meta-Model to a Generic Level for Variability Mining

During our work on lifting our Family Mining algorithms to a generic level, we make use of the common structure between block-based languages (cf. Section 2.1). Independent of their representation (i.e., graphical or textual), block-based models can be perceived on an abstract level as *graphs* that consist of *nodes* with *edges* connecting them. Depending on the used block-based language, nodes might also enable hierarchical modeling using *hierarchy containers*. These nodes and edges allow modeling the behavior of systems and can have different degrees of relevance for the execution. In addition, special *execution start nodes* allow to indicate where the model's execution is started. In certain languages only the nodes influence the execution, while edges only allow transition and passing of data between them (e.g., Matlab/Simulink or IEC 61131-3 FBDs [Int13]). In other languages the edges have a much higher influence as they model behavior executed when transitioning between nodes (e.g., statecharts).

By implementing all our Family Mining algorithms on this level of abstraction, we allow traversal of the analyzed models without knowing their actual details. The concrete comparisons during the traversal are realized by passing the abstract elements to the custom-tailored metric defined in *Step 3* of our guidelines (cf. Section 4.3).

To enable the realized generic Family Mining algorithms, developers have to classify the meta-model classes of model elements as execution start nodes, nodes, edges and hierarchy containers. We provide the following two possible ways of doing this. First, we provide four additional classes `ExecutionStartNodeEntity`, `NodeEntity`, `EdgeEntity` and `ContainerEntity` in our base meta-model. By inheriting from these classes, we allow developers to assign the different categories to their meta-model classes. For example, the `State` class from our meta-model in Figure 4.3 inherits from the `NodeEntity` class, while the `Transition` and `Region` class inherit from the `EdgeEntity` and `ContainerEntity` classes, respectively. In addition, these classes already enable variability-aware modeling following Definition 4.3 by inheriting from the `VariableElement` class. Second, we allow developers to use a custom `EAnnotation` to assign one of the four categories by simply adding corresponding annotations to their classes. While using the annotative solution is less invasive than inheriting from our base meta-model and allows more flexibility in cases where direct modification of meta-model classes is not possible, it does not directly provide solutions to annotate identified variability following Definition 4.3. Thus, we encourage inheriting from our base meta-model if possible, because this solution already creates a variability-aware meta-model when additionally considering point one from Definition 4.3 during meta-model design.

4.2.4. Automatic Generation of Meta-Models for Variability Mining

To ease adapting Family Mining for new languages using our provided base meta-model, we developed the textual *Domain Specific Language (DSL)* Vampire[8] enabling developers to describe and generate their meta-models for any new language. While specifying a meta-model using a textual description might not be intuitive for developers in the first place, it provides advantages in later steps of our Family Mining adaptation. Besides enabling us to generate Eclipse plug-ins containing corresponding EMF Ecore meta-models directly taking advantage of our base meta-model, such Vampire DSL descriptions allow us also to generate large parts of the plug-ins needed to extend our generic Family Mining algorithms for new languages.

When designing the Vampire DSL our goal was to minimize the overall effort for developers during adaptation of Family Mining for their languages. Thus, we designed the language to use a small vocabulary and sensible defaults whenever possible. For instance, the language easily allows to define the source and target nodes for created edge classes. In this case, we assume a default cardinality of $[1..1]$ as most languages use edges with distinct single source and target nodes. However, we also allow to override this default by custom cardinalities. Similarly, it is possible to define incoming or outgoing edges for nodes (with default cardinality of $[1..*]$) and sub entities (with default cardinality of $[1..*]$) or parents (with default cardinality of $[1..1]$) for models, containers and nodes. Furthermore, we generate sensible default names for references storing such relations classes (e.g., `parentRegion` for states with a parent relation to regions), which can easily be overwritten with custom names.

In Listing 4.2, we show the Vampire DSL description for the simplified statechart meta-model in Figure 4.3. The `Settings` block contains the minimal information needed to generate valid Ecore files. The `ModelEntity`, `NodeEntity`, `ExecutionStartNodeEntity`, `ContainerEntity`, `EdgeEntity` and `Entity` classes can be used to model language elements that directly map to the corresponding classes in the base meta-model. When using these classes and the provided key-

[8] Variability-Aware Meta-model Purpose-language Introducing Reverse-Engineering

words to express the relations between model elements, we can automatically derive the necessary containment hierarchy that is required for the serialization of model instances using EMF. In case of additional custom classes, this containment hierarchy can be customized for corresponding references. In addition, the `ClassEntity` (not shown in this example) allows to model classes that are not inheriting from any of the base meta-model classes. In fact, this allowed us to define and generate our base meta-model with our Vampire DSL (cf. the Vampire description in Listing A.1 of the appendix). Using the Vampire DSL description in Listing 4.2, it is possible to generate the simplified statechart meta-model in Figure 4.3. Furthermore, this description can be extended in later steps of the Family Mining adaptation to automatically generate further plug-ins comprising necessary adaptations and used comparison metrics. Overall, this reduces the effort for developers to adapt our Family Mining for their purposes.

In the appendix, we show our complete Vampire DSL descriptions for Family Mining of:

- All seven statechart notations supported by our statechart Family Mining (cf. Listing C.1).
- Matlab/Simulink models and IEC 61131-3 FBDs [Int13] (cf. Listing D.1).

Given that a *complete* language analysis was executed and that *suitable* language concepts matching the domains requirements were selected in *Step 1* of our guidelines (cf. Section 4.1), we can assume that a *well-formed* meta-model allows to store *semantically correct* model instances conforming to the analyzed language. By *well-formed* we refer to a meta-model that conforms to the meta-modeling languages constraints and considers all analysis results from *Step 1* of our guidelines (cf. Section 4.1). Overall, we have to keep in mind that modifications of reference cardinalities to create a variability-aware meta-model might allow to create "illegal" model instances (e.g., a statechart with multiple initial states), which is necessary to create 150% models, though (cf. Section 4.2.3). Based on the created meta-model and the classification of the relevance of language elements in *Step 1* of our guidelines (cf. Section 4.1), it is possible to define a metric enabling comparison of these elements.

4.3. Defining a Custom-Tailored Metric

Based on the detailed analysis of language concepts in *Step 1* (cf. Section 4.1) of our guidelines and a corresponding meta-model created in *Step 2* (cf. Section 4.2) of our guidelines, it is possible to execute comparisons between model variants in the corresponding language. To have a clear separation between the algorithms traversing the compared model variants and the logic comparing the corresponding model elements, we introduced metrics as an additional abstraction layer to measure the similarity of model elements. Such metrics are designed to be easily exchangeable for different requirements or languages and enable custom-tailored Family Mining by adjusting the comparison logic to specific needs. In addition they provide a focused view on the actual comparisons without confusing developers with unnecessary details of the model traversal.

4.3.1. Measuring Similarity of Model Elements

According to Fenton et al. [FB14], it is essential to gain a deep understanding of software attributes and corresponding tools to define clear ways of measuring them. For our goal to compare different model elements, this means that we have to understand how to *measure* their *similarity* based on their *characteristics*. Fenton et al. [FB14] define measurement as follows: "*Measurement* is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way

```
Settings {
  modelName stateoriented
  modelUri "http://www.tu-bs.de/isf/familymining/stateoriented"
  modelPrefix stateoriented
  namespacePrefix de.tu_bs.cs.isf.familymining
  fileExtension statechart
}

ModelEntity Statechart subEntities Region rootRegion [1..1]

abstract NodeEntity AbstractState
  parentEntity Region parent incoming Transition
{
  attribute name, EString, [1..1]
}

NodeEntity State subEntities Region regions [0..-1]
  extends AbstractState outgoing Transition

NodeEntity FinalState extends AbstractState

ExecutionStartNodeEntity InitialState extends State

ContainerEntity Region parentEntity AbstractState parent [0..1]
  subEntities InitialState initialStates [1..-1],
    State states, FinalState finalStates
{
  attribute name, EString, [1..1]
}

EdgeEntity Transition sourceEntity AbstractState source
  targetEntity AbstractState target
{
  containment reference label, TransitionLabel, [1..1]
}

Entity TransitionLabel {
  attribute events, EString, [0..-1]
  attribute conditions, EString, [0..-1]
  attribute actions, EString, [0..-1]
}
```

Listing 4.2: Vampire DSL description allowing automatic generation of the simplified statechart metamodel in Figure 4.3.

so as to describe them according to clearly defined rules." Thus, measurement captures details of observed entities (i.e., in our case the compared model elements) by analyzing their attributes (i.e., in our case their properties) and assigning values to them to make them comparable. Following this definition, we analyzed in more detail what constitutes the similarity of model elements.

Definition 4.4: Similarity of Model Elements

The overall *similarity* of two model elements me_1 and me_2 can be calculated based on the *properties* modeling the elements' characteristics and behavior. During the comparison of corresponding property values, we can distinguish between *binary* similarity (e.g., whether two statechart states are initial states or not) and *relative* similarity (e.g., the names of two statechart states are only 60% similar due to a renaming).

When applying this definition to the calculation of similarity values during the Family Mining of model variants, we need a clear understanding of all used language concepts to enable the calculation of sensible similarity values. On the one hand, we have to understand which language elements can be compared with each other. For example, it does not make sense to compare a state with a transition as these language elements represent completely unrelated functionality (i.e., an execution state vs. a transition between such states). While it would make sense to compare an initial state with a normal state as they only have minor differences. On the other hand, it is important to know all relevant element properties to include all crucial characteristics in the calculation of an overall similarity.

In addition, the compared language elements might have a large number of properties that have a partial impact on the elements' behavior and the overall execution of the developed systems. Thus, for a calculation of an overall similarity between compared model elements it makes sense to assign different weights regarding the properties' impact on the similarity of language elements. As a result, we execute comparisons during our Family Mining based on weighted similarity metrics.

Definition 4.5: Weighted Similarity Metrics

A *similarity metric* defines which elements can be compared and how a similarity value can be calculated for two corresponding model elements based on their property values. For each property a rule exists describing how to compare its values and to calculate a similarity value within a clearly defined *interval* (i.e., the properties' similarity can be binary or relative). The resulting property similarities are *weighted* to enable ranking of their importance regarding the overall similarity of the compared model elements. To ensure comparability the overall similarity value for compared model elements should be *normalized* in the interval [0..1] and, thus, the summed up weights should equal 1.

As a result, the detailed analysis of the newly added modeling language in *Step 1* of our guidelines (cf. Section 4.1) not only pays off to build an appropriate meta-model in *Step 2* of our guidelines (cf. Section 4.2), but also provides all necessary details to make such decisions during the definition of concrete metrics.

4.3.2. Defining a User-Adjustable Similarity Metric for Model Elements

As extensibility and adaptability for new languages is one of our main concerns for the Family Mining algorithms, we realized the metric as one of the central artifacts of our Family Mining implementation. To enable developers to easily use different metrics based on the current setting, we allow to easily exchange concrete metrics that are called during traversal of compared models by our generic Family Mining algorithms. These exchangeable metrics execute the actual comparisons to calculate the compared elements' similarity and, thus, allow to incorporate domain knowledge. Based on the metric, all comparisons are represented by so-called comparison elements.

Definition 4.6: Comparison Elements

A *comparison element ce* stores for each comparison the compared two model elements me_1 and me_2 together with a *weighted similarity value* sim_{ce}^{w}.

This similarity value stores the results of all executed comparisons for the model elements' properties together with the applied weightings w and, thus, allows detailed analysis of the calculated overall similarity value sim_{ce}^{w}.

First, developers have to identify all model elements that are later compared during the Family Mining. Generally speaking, this affects all node, edge or container entities from the metamodel. However, in certain cases also other elements referenced by these base model constructs have to be considered. For example, in Figure 4.3 the `Transition` edge entity references the `TransitionLabel` class, which defines the transitions behavior. Based on these results, developers have to select all properties that should be considered during the actual comparison of elements. Normally, these properties overlap with the relevant properties identified during the language analysis in *Step 1* of our guidelines (cf. Section 4.1) and include element *names*, *interfaces* and properties influencing the elements' *behavior*.

To identify sensible weightings for the selected properties of compared model elements, developers have to execute a ranking of the properties' impact on the overall similarity of the compared model elements. As model elements with a high similarity are more likely to have the same functionality, we recommend to assign higher weights to properties with a high impact on model execution behavior (e.g., the name of a state has a much lower impact than the state's incoming transitions triggering the state). According to Fenton et al. [FB14], a deep understanding of the properties' characteristics and their influence on the considered system is needed to execute such a ranking. However, in most cases such a deep understanding is not available and has to be gained by a detailed analysis of the relations between model elements and their impact on the overall similarity. Thus, we emphasize the importance of a thorough language analysis in *Step 1* of our guidelines (cf. Section 4.1) as it can provide such details and enable sensible rankings.

Finding this ranking highly depends on the used language and current context (e.g., the domain or company adapting Family Mining). As a result, there is no definite answer. We can only give general guidelines based on our experience of defining metrics for Family Mining of statecharts, Matlab/Simulink models and IEC 61131-3 FBDs [Int13]. In Figure 4.5, we present a workflow that proved to yield suitable results during adaptation of Family Mining for these languages. It basically consists of four steps:

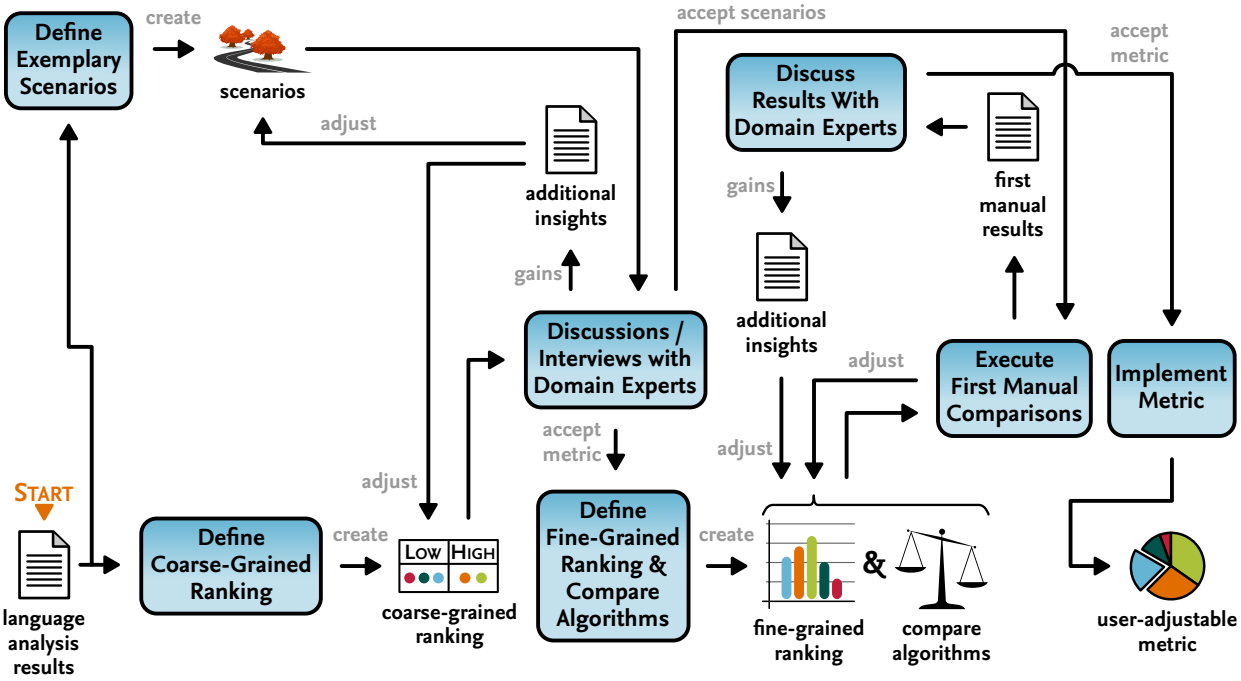


Figure 4.5.: Workflow used to define concrete user-adjustable metrics based on a detailed language analysis.

Defining Exemplary Scenarios As an initial step to approach the definition of a concrete metric, we recommend to develop exemplary scenarios allowing to analyze the impact of different metric settings. These scenarios are later used to execute first manual comparisons with the created metric and serve as basis for discussions. Thus, the developed scenarios should represent small but realistic situations that can occur when comparing models in the considered modeling notation. For example, for statecharts it could be interesting to consider scenarios with two compared states where *none* of them is hierarchical or parallel, *both* are hierarchical or parallel or *only one* of them is hierarchical or parallel. Other scenarios could involve states with the *same* or *differing* numbers of incoming or outgoing transitions. The developed scenarios do not necessarily have to be implemented for the manual comparisons and can also be documented using pen and paper. However, we recommend to actually implement them in the considered modeling tool as the scenarios can also serve as initial tests after implementing the metric and adapting our Family Mining algorithms.

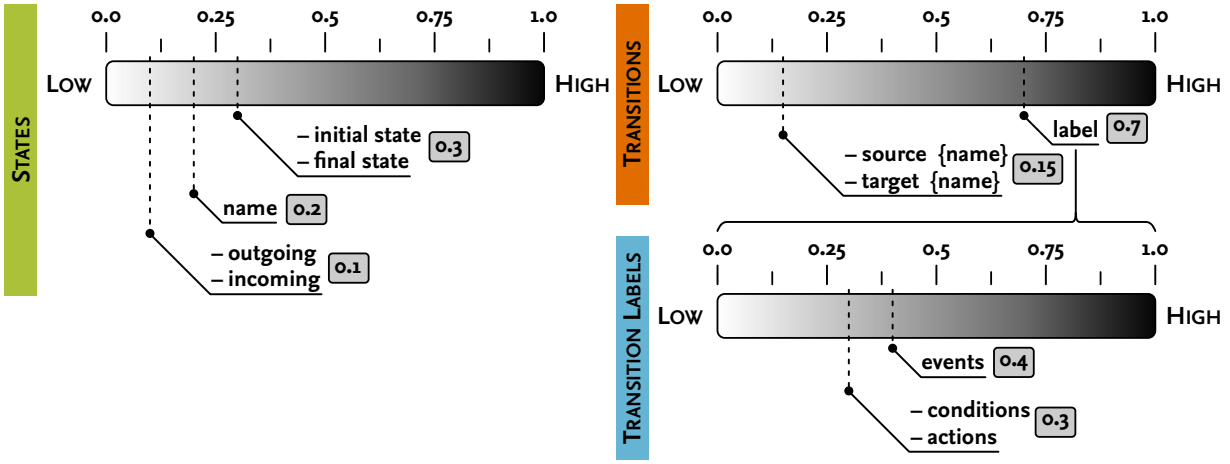
Defining Coarse-Grained Ranking Before realizing a concrete metric with corresponding comparison algorithms, we recommend to define a coarse-grained ranking of impacts on the similarity of compared model elements (i.e., how much the elements' properties influence their similarity). Possible categories for this ranking could be, for example, Low, Middle, High or 1 to 5. The rational of this recommendation is to incrementally refine the categorization prior to defining concrete weights for the different element properties. This allows a detailed discussion of the initial ranking and the created exemplary scenarios with experts to clear misunderstandings by involving their feedback at an early stage of the metric creation. In addition, we observed that (semi-)structured interviews [Pat87] can have a positive effect in cases where already a strong understanding of the language and the impact of different element properties exists. Such interviews are a possible way to present a preliminary perception of element similarity to experts and ask to which degree they

agree with it. Furthermore, questionnaires about the experts' assessment of concrete scenarios can ease the development of specific weights in the next step. Based on the feedback of the experts, the created scenarios and coarse-grained rankings should be adjusted and iterated until realistic scenarios and a clear coarse-grained classification of the impact of different element properties exists.

Defining Fine-Grained Ranking While it can be sensible to define a "global" coarse-grained ranking of all element properties (e.g., states, transitions and regions for statecharts) to get an overview of their overall impact on the compared models, it is imperative to execute the fine-grained ranking for each model element separately. This is essential as not all language elements can be compared with each other (cf. Section 4.3.1). Thus, after accepting the previous coarse-grained ranking, developers have to define a separate fine-grained ranking for the properties of each model element selected for comparison. For this ranking, developers should keep Definition 4.5 in mind and have to consider especially the normalization of calculated similarity values (i.e., for each model element, all summed up weights should sum up to 1). From our experience, we recommend developers to also discuss the thresholds used to categorize the identified variability during the merging of 150% models (cf. Section 4.6) at this point. By knowing these thresholds in advance, it is possible to select all weights in a way that, for example, minor differences do not directly change the variability from mandatory to alternative. Furthermore, severe changes can be penalized by strongly reducing the calculated similarity value.

In Figure 4.6, we show an exemplary ranking with concrete weights for states, transitions and their labels of the simplified statechart meta-model in Figure 4.3. For each of the model elements, we show a separate scale in the interval $[0..1]$ together with markers and gray boxes showing the assigned weights. For multiple properties with the same weight (e.g., the incoming and outgoing transitions of states) only a single marker and single gray box exist. For states, we regard the interfaces as the lowest indicator for similarity as the states execution is triggered but not defined by them. Thus, we rank the incoming and outgoing transitions with a weight of 0.1 each. The compared names have a slightly higher impact as they make the executed states distinguishable (i.e., we rank them with a weight of 0.2). Furthermore, we regard properties influencing the execution (i.e., whether the compared states are marked as initial or final states) as the most important factors during the comparison of states (i.e., we rank them with a weight of 0.3 each). In our example, the metric for transitions represents a special case, as we compare the corresponding source and target states based on their corresponding names and consider matching source and target states with a weight of 0.15 each. For other languages it could make sense to include additional details in such comparisons. For example, in Matlab/Simulink, it could be possible to check whether the source and target blocks of a connector not only have the same name, but realize the same functionality based on their type. In addition to the transitions' source and target states, we consider their labels with a weight of 0.7 and compare them using a sub metric with weights for the labels' events (i.e., a weight of 0.4), conditions (i.e., a weight of 0.3) and actions (i.e., a weight of 0.3).

Defining Comparison Algorithms for Compared Properties In addition to assigning concrete weights to compared properties, developers should define clear algorithms on how to compare the properties. In accordance with Definition 4.4, the corresponding algorithms either have to return a binary similarity (i.e., whether the compared properties are equal) or a relative similarity showing the degree of similarity. For our exemplary statechart metric in Figure 4.6, we use both comparisons.



(a) Ranking for statechart states.

(b) Ranking for statechart transitions and their labels.

Figure 4.6.: Ranking for the properties of states, transitions and transition labels for the simplified statechart meta-model in Figure 4.3.

For instance, binary similarity comparisons are applied to check whether compared states are marked as initial or final states. Relative similarity comparisons are executed, for example, to identify how similar the number of incoming and outgoing transitions is or how many of the events, conditions and actions are matching for the transitions' labels. A special case represent all string comparisons (e.g., the names of states) as it is possible to either identify binary similarities (i.e., by checking whether the strings are 100% equal) or calculate relative similarities (e.g., by using the Levenshtein Distance algorithm [Lev66] returning the edit difference between the compared strings). For our realization, we calculate the relative similarity for the comparison of all element names to account for minor deviations caused be renaming these elements during clone-and-own.

Depending on the scenario it can be sensible to apply more complex algorithms to identify the similarity of specific properties. For example, it can be helpful to integrate information from other sources (e.g., an external database) to enrich the considered details about compared elements and improve the comparison results or to apply more complex algorithms (e.g., to compare code artifacts). For example, we currently use binary similarity comparisons to compare code artifacts (e.g., used to model the events of transitions) by applying a simple equality check of the corresponding strings. However, it can make sense to apply more complex algorithms to increase the accuracy of the similarity calculation algorithms in cases where this simple approach fails to provide sensible results (e.g., inaccurate variability relations are identified). By parsing the compared code artifacts to an *Abstract Syntax Tree (AST)* – i.e., an abstract tree representation of the syntactic code structure – or corresponding meta-model representation and comparing them based on this representation it is possible to identify a more fine-grained similarity between the code artifacts. For instance, developers could adapt our algorithms from [WTS+16] that we developed to compare object-oriented source code based on model-based representations. A sensible scenario could be *Function Blocks* in Matlab/Simulink whose functionality can be defined using C or C++ code artifacts. Comparing such potentially long source code artifacts line by line is not a promising approach as lines could have been added in between other lines or complete blocks of code could have been rearranged. However, it is important to keep in mind that such techniques increase the comparison complex-

ity and, thus, the overall runtime. As a result, we decided to keep our applied statechart metric as simple as possible and only add such complex algorithms when identifying inaccurate variability relations. During the evaluation of our results, we were able to show for the statechart variants from our current case study that in this setting such additional algorithms are not necessary. Thus, we recommend developers to keep their metrics as simple as possible and only add additional and more complex comparison logic if necessary.

Based on the assigned weights and selected comparison algorithms it is now possible to compare the corresponding language elements. In Equation 4.1, we show an exemplary similarity calculation for a comparison of the `cls_unlock` states from the two CLS variants with differing PWs from our running example in Figure 2.5 using the metric defined in Figure 4.6a. In this example, the Levenshtein Distance algorithm returns a 100% similarity for the compared state names. Both binary similarity checks, whether the compared states are initial states and final states, return a 100% similarity as these properties match (i.e., they both are initial states and both are *not* final states). The only difference for the compared states affects their outgoing interfaces as the `cls_unlock` state in the CLS ManPW variant has two outgoing transitions in contrast to the single outgoing transition for the `cls_unlock` state from the CLS AutoPW variant (i.e., reflected by the relative similarity of $\frac{1}{2}$). Overall, the compared states have a similarity of 95% according to the applied metric.

$$sim^{w}_{ce} = \underbrace{\frac{1}{1} \times 0.1}_{\text{incoming transitions}} + \underbrace{\frac{1}{2} \times 0.1}_{\text{outgoing transitions}} + \underbrace{1 \times 0.2}_{\text{name}} + \underbrace{1 \times 0.3}_{\text{initial state}} + \underbrace{1 \times 0.3}_{\text{final state}} = 0.95 \tag{4.1}$$

To evaluate the defined weights, developers should use the previously created exemplary scenarios to manually execute first comparisons in realistic scenarios. These results should be iteratively discussed with domain experts and be used to gradually improve the defined weights. Only after getting satisfactory results from these manual comparisons and corresponding feedback from the experts, developers should consider implementing the metric for application in our Family Mining algorithms.

In the appendix, we show our complete metrics used for Family Mining of:

- All seven statechart notations supported by our statechart Family Mining (cf. Table C.1).
- Matlab/Simulink models and IEC 61131-3 FBDs [Int13] (cf. Table D.1).

4.3.3. Automatic Generation of Similarity Metrics

While developers can manually implement their metric using the provided extension point, we also allow generation of such metrics by extending meta-model descriptions in our Vampire DSL. Using an additional `Weights` keyword it is possible to define concrete weights for the properties of elements and details on how to compare them. Similar to the basic Vampire DSL, we use sensible defaults wherever possible (e.g., we compare strings based on the Levenshtein Distance algorithm by default). To compare different properties, we provide the following means to allow users to control the generated comparison algorithms:

- Users can define for each string property whether they want to compare it using the Levenshtein Distance algorithm (i.e., the default solution) or the Java equals method (i.e., using the `equals` keyword).

- Users can generate interface comparisons based on the number of incoming and outgoing edges (i.e., the interfaces' size) using the `incoming` and `outgoing` keywords.
- Users can check whether model elements have a specific type (e.g., whether a state represents an initial state) using the `typeCheck` keyword.
- Users can define which properties of edges' source and target nodes should be compared (e.g., their names) using the `sourceEntity` and `targetEntity` keywords.
- Users can redirect the comparison of a property to a sub metric (e.g., to compare the labels of transitions in our simplified meta-model) using the `redirect` keyword.
- Users can use the calculated similarity of sub entities to calculate the similarity of compared elements (e.g., the similarity of a statechart region can be calculated based on the similarity of its sub states and transitions) using the `subEntities` keyword.

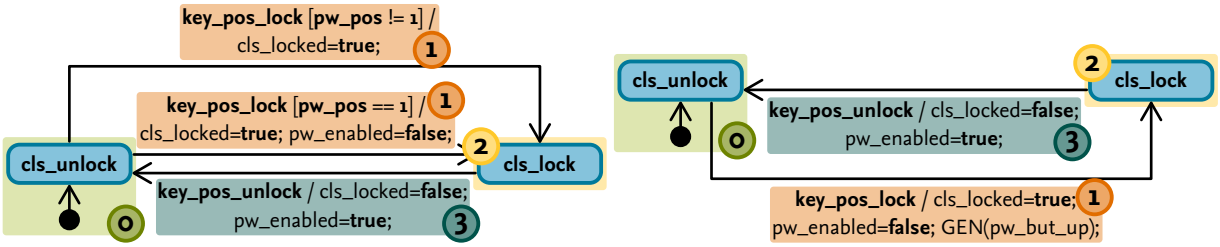
Based on corresponding Vampire DSL descriptions, we are able to generate weighted metrics for all specified model elements together with concrete algorithms to compare interfaces, types, single property values and list property values. In Listing 4.3, we show the extension of our Vampire meta-model description in Listing 4.2 for our simplified statechart meta-model in Figure 4.3. Here, we were able to decompose the overall metric for states into comparisons of properties concerning the corresponding hierarchy element (e.g., the type checks only concern the corresponding classes `InitialState` and `FinalState`). We could have added each of the comparison descriptions to the `Weights` of the corresponding parent class `AbstractState`. However, we realized the possibility of inheritance to allow users for a more flexible and modular design with complex and large metrics by generating corresponding metric classes with a matching inheritance structure. The generated metrics observe inheritance rules defined in the meta-model description and, thus, a sensible decomposition of the metric is possible and allows an easier analyzes of the generated metrics. When executing the comparison for any class inheriting from `AbstractState`, all decomposed comparisons (i.e., in the example the `AbstractState`, `State`, `InitialState` and `FinalState` comparisons) are executed by method calls to calculate the overall similarity.

In addition to the definition of a concrete metric using our Vampire DSL, we allow to use `EAnnotations` in cases where an existing meta-model is used. By parsing the annotations from the meta-model file, we are able to provide and generate the same metrics as with corresponding Vampire DSL descriptions. In Table B.1 of Appendix B, we summarize all possible meta-model annotations together with a mapping to their corresponding Vampire DSL keywords. Overall, both the extended Vampire DSL and the provided `EAnnotations` allow to largely reduce the adaptation effort for our Family Mining as they allow users to easily define concrete metrics for their meta-models to generate corresponding Family Mining plug-ins. The generated metrics can also be custom-tailored towards the current settings by a) adjusting the selected weights using the integration of the metric in our Family Mining *Graphical User Interface (GUI)* and b) implementing custom comparison algorithms in situations where additional logic is needed for comparisons of elements.

Based on the either manually implemented or automatically generated metric for the new language, it is now possible to compare model elements with each other. By passing model elements that should be compared to such metrics, our generic Family Mining algorithms are now capable of executing concrete comparisons and creating comparison elements with similarity values.

```
// ...

Weights AbstractState {
  name 0.2
  incoming 0.1
}

Weights State {
  outgoing 0.1
}

Weights InitialState {
  typeCheck InitialState 0.3
}

Weights FinalState {
  typeCheck FinalState 0.3
}

Weights Region {
  subEntities AbstractState 0.5
  subEntities Transition 0.5
}

Weights Transition {
  redirect label 0.7
  sourceEntity name 0.15
  targetEntity name 0.15
}

Weights TransitionLabel {
  events 0.4
  conditions 0.3
  actions 0.3
}
```

Listing 4.3: Vampire DSL description allowing automatic generation of a concrete similarity metric for the simplified statechart meta-model in Figure 4.3.



(a) With a Manual Power Window (ManPW). (b) With an Automatic Power Window (AutoPW).

Figure 4.7.: Stages annotated to the two running example variants of a car Central Locking System (CLS) with differing Power Windows (PWs) in Figure 2.5.

4.4. Comparing Model Variants

After defining a user-adjustable metric in *Step 3* of our guidelines (cf. Section 4.3) to allow comparison of all relevant language elements based on their attributes, it is now possible to execute our generic Family Mining comparison algorithm to identify possible variability relations between model variants. In this section, we explain the details of our comparison algorithm and how we exploit the general structure of block-based languages (i.e., their nodes, edges and containers) to allow generic comparison of model variants that were defined using meta-models extending our base meta-model classes or the corresponding `EAnnotations` (cf. Section 4.2.3).

Our comparison algorithm is realized as an iterative *pairwise* algorithm and, thus, compares two model variants at a time. The general idea of the algorithm is to analyze the models' execution-flow from the beginning to the end and to compare model elements along this path. Thus, we refer to the proposed comparison algorithm as the Execution-Flow Analysis (EFA) algorithm. By virtually dividing the analyzed models, the algorithm exploits the structure of block-based languages (i.e., their realization as graphs).

Definition 4.7: Stages

Stages virtually separate compared models or model hierarchies. Each stage st only contains model elements that have the same *distance* d_{st} on the shortest path from the execution start (e.g., the initial states of a statechart) to the corresponding model elements.

In Figure 4.7, we annotated and highlighted the stages for the model variants from our running example in Figure 2.5. As we can see, the `cls_unlock` initial states are part of stage st_0 as they have a distance $d_{st} = 0$ to the execution start. Similarly, the outgoing transitions of the `cls_unlock` states form stage st_1 with distance $d_{st} = 1$ and so on. Model elements from compared model variants forming stages with the same distance d_{st} are compared by creating all possible combinations (e.g., the transitions in stage st_1 from Figure 4.7 create two possible combinations).

To consider different hierarchy levels during the comparison of hierarchical models, we rely on a *Depth-First Search (DFS)*[9] strategy to generate corresponding comparisons. The EFA algorithm is capable of comparing more than two models by allowing 150% models, generated by the merge step, as

[9]The DFS algorithm was widely acknowledged as an efficient way of solving different graph problems after it was published by Tarjan [Tar71] and Hopcroft et al. [HT73]. However, it was already used since the 19th century as *Trémeaux's Algorithm* (cf. [Luc82]) to efficiently solve mazes [Eve11].

input for comparisons with further variants. This way, we are able to iteratively compare and merge large numbers of model variants to identify their fine-grained variability relations. While we were expecting that this iterative merging approach would yield different results for varying input orders of model variants, we did not encounter such situations during our evaluation and corresponding experiments. Although we did not execute a formal proof, we believe that this is actually due to the iterative merging, because it allows the Family Mining to always consider all model elements from previous comparisons. As a result, there should not be any differences between executions with varying input orders as all relevant information of elements are stored in the merged 150% model to not loose any details in subsequent comparisons.

Prior to starting the comparison of a set of input model variants, the EFA algorithm identifies the initial two model variants. We distinguish between two types of model variants.

Definition 4.8: Base Model Variant and Comparison Model Variants

The *base model variant* mv_{base} serves as basis for all comparisons with the remaining *comparison model variants* $mv_{comparison}$ and is used to iteratively merge all identified variability relations.

To select the base model variant, we allow users to apply their domain knowledge by manual selecting a sensible variant. In addition, we implemented an automatic strategy to select the base model variant based on the models' size. By selecting the smallest model from a set of input models, we attempt to automatically select the core of the analyzed variants as we argue that clone-and-own approaches are executed in an additive manner (i.e., only additional behavior is added) and it is less likely that behavior is removed. Overall, the selection of the base model variant does not have an impact on the executed Family Mining as the iterative comparison of all input variants ensures that all variability information is considered in a created 150% model. However, we use the base model variant as the core variant during a migration of the compared variants from a corresponding 150% model to an SPL (cf. Chapter 5).

4.4.1. Executing the Execution-Flow Analysis Comparison Algorithm

In Algorithm 4.1, we depict the `compareNextHierarchyLevel` method, which is called when starting the comparison between two model variants. Furthermore, this method is recursively called for each hierarchy level identified for entities $entity_{base}$ and $entity_{comparison}$ throughout the comparisons. The method identifies all sub entities of the passed entities and executes the EFA algorithm for them. First, the method identifies all sub node entities N_{base} and $N_{comparison}$ as well as all sub container entities C_{base} and $C_{comparison}$ (cf. Line 2 – 5) by analyzing the types of the sub entities. Here, the generic meta-model basis introduced during the creation of the meta-model in *Step* 2 of our guidelines (cf. Section 4.2.3) takes effect as inheriting from our base meta-model or using corresponding `EAnnotations` allows to categorize all identified sub entities. In case of sub node entities, the algorithm first identifies all execution start node entities N^{es}_{base} and $N^{es}_{comparison}$ forming stage st_0 in the input lists by calling the `identifyExecutionStartNodes` method (cf. Line 9 – 10). These execution start nodes serve as the initial node entities that are compared and from which point the EFA algorithm continues the execution-flow based comparisons. Similar to the categorization of node and container entities, this method exploits the information about the entities' type to identify all execution start node entities. In case none of the node entities in the used meta-model was

Input: two entities $entity_{base}$ and $entity_{comparison}$,
initial input $mv_{base} \in MV$ and $mv_{comparison} \in MV$ with $mv_{base} \neq mv_{comparison}$
Output: a list of possible comparison elements CE_p representing the comparison of the entities' sub elements

```
1  method compareNextHierarchyLevel(entity_base, entity_comparison) : CE_p is
2      N_base ← subNodes(entity_base)
3      N_comparison ← subNodes(entity_comparison)
4      C_base ← subContainers(entity_base)
5      C_comparison ← subContainers(entity_comparison)
6      if C_base ≠ ∅ OR C_base ≠ ∅ then
7          CE_p ← compareContainers(C_base, C_comparison)
8      else
9          N^es_base ← identifyExecutionStartNodes(N_base)
10         N^es_comparison ← identifyExecutionStartNodes(N_comparison)
11         CE_p ← compareNodes(N^es_base, N^es_comparison)
12     end
13     return CE_p
14 end
```

ll. 2 – 5: Identify sub nodes / sub containers in the entity hierarchy.

ll. 9 & 10: Identified either based on type or based on custom implementation.

Algorithm 4.1: The method triggering the Execution-Flow Analysis (EFA) comparison algorithm.

categorized as an execution start node entity, the user has to implement a custom algorithm to select sensible starting points for the comparisons. For example, our meta-model for Family Mining of Matlab/Simulink models and FBDs [Int13] (cf. Figure D.1) only contains a generic `Block` node entity and the blocks' types are specified by assigning matching strings. In this case, we realized a custom implementation of the `identifyExecutionStartNodes` method returning blocks with sensible types (e.g., *Constant Blocks* or *Inport Blocks*). In contrast, for statecharts, we are able to automatically identify all initial states based on the meta-model inheritance structure (i.e., initial states are marked as execution start node entities).

For industrial settings, it is important to keep in mind that a clear identification of distinct execution start nodes is not always possible. For example, it is a common strategy to use MIL strategies to test and simulate models in scenarios where the developed model is dependent on external factors (e.g., the environment of a car) that either cannot be directly used for model testing or can only be used involving large costs making the tests inefficient (e.g., in case of specialized expensive hardware) [Plu06]. In such scenarios often special models are used to simulate these external factors allowing developers to influence them and the way their tested model is triggered. In Figure 4.8, we show the highest hierarchy level of an exemplary Matlab/Simulink model that realizes a control circuit. Here, the `Environment` subsystem models the external factors (e.g., the environment of a car) influencing the `Control Algorithms` implemented by the developers. Although this circuit represents a legal Matlab/Simulink model, it does not provide distinct execution start nodes (e.g., *Inport blocks*) on the highest hierarchy level and, thus, prevents the execution of our algorithm. To cope with such situations in a project with one of our industry partners, we applied

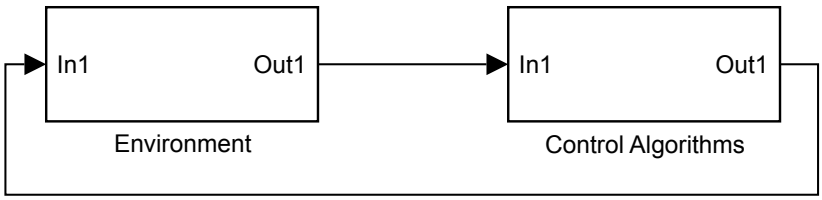


Figure 4.8.: Exemplary MIL control circuit in Matlab/Simulink.

a simple name-based selection algorithm that compares the names of blocks and randomly selects two blocks with equal names from the base model variant and the comparison model variant. While this solution obviously fails in situations where no blocks with equal names can be identified, the probability is considerably low. Developers from our industry partner confirmed that the highest hierarchy of such models are normally designed in the same way due to conventions in the companies' development guidelines. Thus, the block names and the layout of such control circuits are basically the same in all models. In addition, when comparing such models in clone-and-own scenarios the probability of changed names is even lower as any changes to the copied models are most likely applied to the `Control Algorithms` subsystem comprising the developed algorithms. As a fall-back solution it is possible to provide an interface for a to manual selection of entry points by developers for such control circuits.

4.4.2. Comparing Model Nodes

After identifying the execution start nodes, the algorithm continues by calling the `compareNodes` method to compare these nodes forming stage st_0 (cf. Line 11). In Algorithm 4.2, we depict the basic algorithm for this method. This algorithm compares each node from N_{base} with each node from $N_{comparison}$ (cf. Line 4 – 12) by calling the applied metric to create corresponding comparison elements ce_p (cf. Line 6). For each comparison, the algorithm recursively calls the `compareNext-HierarchyLevel` method (cf. Algorithm 4.1) passing the currently compared nodes $n_{base} \in N_{base}$ and $n_{comparison} \in N_{comparison}$ to create comparison elements CE_p^{sub} for any entities that might exist in the nodes' hierarchy. For these possible sub comparison elements, the matching algorithm is executed to identify distinct comparison elements CE_m^{sub} between the hierarchy entities (cf. Section 4.5) and to store them as sub comparison elements of ce_p (cf. Line 7). As the overall similarity of hierarchical comparison elements depends on the similarity of their compared properties, but also on the similarity of their sub entities, we allow recalculation of the overall similarity by passing the comparison element ce_p back to the metric for corresponding adjustments of the calculated similarity $sim_{ce_p}^{w}$ (cf. Line 9). In case one of the lists N_{base} or $N_{comparison}$ is empty, the algorithm compares the nodes from the non-empty list with `null` (depicted by the empty set $\emptyset$) indicating that they do not have a counterpart in the other model (cf. Line 14 – 18 and Line 20 – 24, respectively). In this case, only one of the models can contain model elements in the hierarchy (i.e., the one with the non-empty nodes list). Thus, the matching algorithm is not called, because the created sub comparison elements directly represent distinct relations for the hierarchy. Furthermore, the similarity recalculation is not executed as the similarity of comparisons without counterparts is 0% and the similarity of hierarchical comparison elements without counterpart does not influence their parent comparison element.

Input: two lists N_{base} and $N_{comparison}$ containing nodes
Output: a list of possible comparison elements CE_p for the compared nodes

```
1  method compareNodes(N_base, N_comparison) : CE_p is
2      CE_p ← ∅
3      if N_base ≠ ∅ AND N_comparison ≠ ∅ then
4          foreach n_base ∈ N_base do
5              foreach n_comparison ∈ N_comparison do
6                  ce_p ← metricComparison(n_base, n_comparison)
7                  CE_p^sub ← compareNextHierarchyLevel(n_base, n_comparison)
8                  ce_p.CE_m^sub ← match(CE_p^sub)
9                  recalculateHierarchySimilarity(ce_p)
10                 CE_p ← CE_p ⋃ ce_p
11             end
12         end
13     else if N_base ≠ ∅ then
14         foreach n_base ∈ N_base do
15             ce_p ← metricComparison(n_base, ∅)
16             ce_p.CE_m^sub ← compareNextHierarchyLevel(n_base, ∅)
17             CE_p ← CE_p ⋃ ce_p
18         end
19     else if N_comparison ≠ ∅ then
20         foreach n_comparison ∈ N_comparison do
21             ce_p ← metricComparison(∅, n_comparison)
22             ce_p.CE_m^sub ← compareNextHierarchyLevel(∅, n_comparison)
23             CE_p ← CE_p ⋃ ce_p
24         end
25     end
26     E_base ← identifySuccessorEdges(N_base)
27     E_comparison ← identifySuccessorEdges(N_comparison)
28     if E_base ≠ ∅ OR E_comparison ≠ ∅ then
29         CE_p ← CE_p ⋃ compareEdges(E_base, E_comparison)
30     end
31     return CE_p
32 end
```

ll. 6, 15 & 21: Call of the metric extension point to compare the nodes.

ll. 7, 16 & 22: Compare next hierarchy level.

l. 8: Store distinct matching of next hierarchy level.

l. 9: Recalculate similarity value based on hierarchy.

ll. 26 & 27: Identify successor edges to traverse the model.

Algorithm 4.2: The Family Mining comparison algorithm for nodes.

4.4.3. Comparing Model Edges

After creating the comparison elements for the node entities of the current stage, the algorithm identifies the successor edges of the subsequent stage and calls the `compareEdges` method to trigger their comparison (cf. Line 26 – 30).

In Algorithm 4.3, we depict the basic algorithm for this method. The general approach is similar to the `compareNodes` method as the same ideas for comparing edges are applied. However, in general edges do not contribute to the functionality of block-based languages. Thus, they might not need to be represented by corresponding comparison elements. For example, in case of Matlab/Simulink models, we did not define a metric for their connectors as they only contribute few details to the similarity of the models (i.e., they only contain an optional name). Thus, we allow developers to decide whether edges actually contribute to the model similarity and allow to deactivate the creation of edge comparison elements (cf. Line 3). Furthermore, as edges do not realize hierarchical structures in block-based languages, we only call the metric to execute the edge comparison. For both, the comparison of nodes and edges, it is important to keep track of elements that were already compared (e.g., during the `metricComparison` method calls). Otherwise, the comparison algorithm could end in an infinite loop for block-based languages allowing *self-loops* (i.e., edges having the same source and target node as in statecharts or Matlab/Simulink models).

4.4.4. Comparing Model Hierarchy Containers

In case the `compareNextHierarchyLevel` method identified container entities as sub elements of the passed entities, the algorithm executes the `compareContainers` method (cf. Line 7). In Algorithm 4.4, we depict the basic algorithm for this method. The general approach is similar to the `compareNodes` method as the same ideas are applied to create the corresponding comparison elements. However, in case of the containers, the algorithm does not have to calculate and compare subsequent stages but only to trigger the comparison of the next hierarchy levels during the creation of the comparison elements.

As mentioned, the described EFA algorithm is capable of iteratively comparing sets of model variants as they can also be executed on 150% models merged during the comparison of two models. Thus, in case of $n > 2$ compared models, the algorithm requires that the identified variability information is merged into legal 150% models following the details in Section 4.6. This way, the EFA algorithm is capable of traversing the 150% models and considering any identified alternative paths in the compared models (e.g., due to alternative edges).

In Table 4.4, we summarize the comparison elements created by the EFA algorithm. For each stage st, we show the created comparison elements ce with their contained model elements from the compared model variants mv_{base} and $mv_{comparison}$ and the similarity values sim^{w}_{ce} calculated according to our simplified statechart metric (cf. Figure 4.6). As we can see, the EFA algorithm identified five possible comparison elements in four stages for the compared model variants from our running example (cf. Figure 2.5). The algorithm identified for both `cls_unlock` and `cls_lock` states a similarity of 95% due to the minor differences of their interfaces in both variants (i.e., the `cls_unlock` and `cls_lock` states in Figure 2.5a have an additional outgoing / incoming transition compared to Figure 2.5b). Furthermore, the transitions in stage st_3 have a similarity of 100%. The remaining comparison elements in stage st_1 both have a lower similarity as differences exist between their guards and actions.

Input: two lists E_{base} and $E_{comparison}$ containing edges
Output: a list of possible comparison elements CE_p for the compared edges

```
1  method compareEdges(E_base, E_comparison) : CE_p is
2      CE_p ← ∅
3      if compareEdgesIsActivated() then
4          if E_base ≠ ∅ AND E_comparison ≠ ∅ then
5              foreach e_base ∈ E_base do
6                  foreach e_comparison ∈ E_comparison do
7                      ce_p ← metricComparison(e_base, e_comparison)
8                      CE_p ← CE_p ⋃ ce_p
9                  end
10             end
11         else if E_base ≠ ∅ then
12             foreach e_base ∈ E_base do
13                 ce_p ← metricComparison(e_base, ∅)
14                 CE_p ← CE_p ⋃ ce_p
15             end
16         else if E_comparison ≠ ∅ then
17             foreach e_comparison ∈ E_comparison do
18                 ce_p ← metricComparison(∅, e_comparison)
19                 CE_p ← CE_p ⋃ ce_p
20             end
21         end
22     end
23     N_base ← identifySuccessorNodes(E_base)
24     N_comparison ← identifySuccessorNodes(E_comparison)
25     if N_base ≠ ∅ OR N_comparison ≠ ∅ then
26         CE_p ← CE_p ⋃ compareNodes(N_base, N_comparison)
27     end
28     return CE_p
29 end
```

l. 3: Comparison elements are only created if the user activated the comparison of edges.

ll. 7, 13 & 18: Call of the metric extension point to compare the edges.

ll. 23 & 24: Identify successor nodes to traverse the model.

Algorithm 4.3: The Family Mining comparison algorithm for edges.

Input: two lists C_{base} and $C_{comparison}$ containing containers
Output: a list of possible comparison elements CE_p for the compared containers

```
1  method compareContainers(C_base, C_comparison) : CE_p is
2      CE_p ← ∅
3      if C_base ≠ ∅ AND C_comparison ≠ ∅ then
4          foreach c_base ∈ C_base do
5              foreach c_compare ∈ C_comparison do
6                  ce_p ← metricComparison(c_base, c_comparison)
7                  CE_p^sub ← compareNextHierarchyLevel(c_base, c_comparison)
8                  ce_p.CE_m^sub ← match(CE_p^sub)
9                  recalculateHierarchySimilarity(ce_p)
10                 CE_p ← CE_p ⋃ ce_p
11             end
12         end
13     else if C_base ≠ ∅ then
14         foreach c_base ∈ C_base do
15             ce_p ← metricComparison(c_base, ∅)
16             ce_p.CE_m^sub ← compareNextHierarchyLevel(c_base, ∅)
17             CE_p ← CE_p ⋃ ce_p
18         end
19     else if C_comparison ≠ ∅ then
20         foreach c_comparison ∈ C_comparison do
21             ce_p ← metricComparison(∅, c_comparison)
22             ce_p.CE_m^sub ← compareNextHierarchyLevel(∅, c_comparison)
23             CE_p ← CE_p ⋃ ce_p
24         end
25     end
26     return CE_p
27 end
```

ll. 6, 15 & 21: Call of the metric extension point to compare the containers.

ll. 7, 16 & 22: Compare next hierarchy level.

l. 8: Store distinct matching of next hierarchy level.

l. 9: Recalculate similarity value based on hierarchy.

Algorithm 4.4: The Family Mining comparison algorithm for containers.

Model Variant	Contents of Comparison Element *ce*	Similarity Value sim^{w}_{ce}
Stage st_0		
mv_{base}	`cls_unlock`	95%
$mv_{comparison}$	`cls_unlock`	
Stage st_1		
mv_{base}	`key_pos_lock [pw_pos != 1] / cls_locked=true;`	65%
$mv_{comparison}$	`key_pos_lock / cls_locked=true; pw_enabled=false; GEN(pw_but_up);`	
mv_{base}	`key_pos_lock [pw_pos == 1] / cls_locked=true; pw_enabled=false;`	72%
$mv_{comparison}$	`key_pos_lock / cls_locked=true; pw_enabled=false; GEN(pw_but_up);`	
Stage st_2		
mv_{base}	`cls_lock`	95%
$mv_{comparison}$	`cls_lock`	
Stage st_3		
mv_{base}	`key_pos_unlock / cls_locked=false; pw_enabled=true;`	100%
$mv_{comparison}$	`key_pos_unlock / cls_locked=false; pw_enabled=true;`	

Table 4.4.: Comparison elements generated by the EFA algorithm for the running example in Figure 2.5, where the annotated stages are highlighted in Figure 4.7.

Overall, the described EFA algorithm allows to execute generic Family Mining on models that were created using meta-models that inherit from our base meta-model or use corresponding `EAnnotations`. By switching between the EFA algorithm enabling the generic traversal of models and concrete comparison algorithms provided by the custom-tailored metric, we enable an almost language-agnostic comparison of model variants. This way, we allow developers to adapt our Family Mining for their block-based languages with low effort as only few methods have to be implemented by them (i.e., the method to identify successor edges for a list of nodes, a method to identify target nodes of edges and, in case no distinct execution start nodes exist, the method to identify sensible start nodes). To ease the adaptation even more, we use the created Vampire DSL descriptions or corresponding meta-model `EAnnotations` to automatically generate all method implementations if possible. For example, in case, no distinct executions start nodes were modeled, we cannot automatically derive the corresponding implementation, but generate a method stub with comments on what the developer has to implement.

4.5. Matching Relations between Model Variants

Usually, the comparison elements created during the traversal of the compared models using our EFA algorithm (cf. Section 4.4) are ambiguous. For example, when comparing the stages st_1 from our running example (cf. Figure 2.5), we create two comparison elements for the compared transitions (cf. Table 4.4). However, to unambiguously merge the identified variability in a 150% model, we need to identify distinct relations between the compared model variants. Thus, we developed a Similarity Based Matching (SBM) algorithm to efficiently remove such ambiguities by analyzing all created possible comparison elements CE_p based on their similarity values. The created SBM algorithm is completely language-agnostic, because it solely operates on the comparison elements encapsulating the compared model elements and the corresponding similarity values without caring about the underlying language details.

The general idea of the SBM algorithm is to identify for each possible comparison element $ce_p \in CE_p$ whether other comparison elements exist that contain the same entity from the base model $ce_p.base$ or the comparison model $ce_p.comparison$. After identifying the comparison elements related with ce_p, the algorithm checks whether ce_p has the highest similarity and matches it. When matching a comparison element, all other related comparison elements are removed from CE_p. By keeping track of all model elements contained in matched comparison elements, the algorithm is capable of identifying model elements that are optional and creates corresponding comparison elements. In case, all related comparison elements have the same similarity, they are marked as ambiguous and sorted to the end of CE_p. By matching other unambiguous comparison elements first, we try to resolve such conflicts. However, in certain situations, we are not able to resolve ambiguity with this strategy. To resolve such deadlocks, we execute decision wizards.

Definition 4.9: Decision Wizards

Decision wizards allow to resolve conflicts during the matching of comparison elements. We distinguish between decision wizards providing *manual* and *automatic* resolution strategies to resolve deadlocks. Manual decision wizards allow domain experts to select matching comparison elements through a GUI. Automatic decision wizards apply resolution algorithms defined prior executing the Family Mining without manual interaction.

4.5.1. Executing the Similarity Based Matching Algorithm

In Algorithm 4.5, we depict the `match` method, which is called upon starting the matching of the created comparison elements. This method initializes the SBM algorithm by storing the passed possible comparison elements CE_p in a global list and creating global lists for the matched comparison elements CE_m. Furthermore, lists of optional entities O_{base} and $O_{comparison}$ for the base model and comparison model are initialized by processing all possible comparison elements $ce_p \in CE_p$ and adding the entities $ce_p.base$ and $ce_p.comparison$ to them (cf. Line 4 – 13). This way all entities are regarded as optional prior to the actual matching. By executing the subsequent matching, we then refine these variability relations by removing all entities that were matched from the lists. Thus, at the end of the matching, the O_{base} and $O_{comparison}$ lists only comprise the remaining optional entities. After initializing the SBM, the algorithm calls the `doMatching` method (cf. Line 14), which is the central matching method that is called until all comparison elements were matched.

Input: a list of possible comparison elements CE_p
Output: a list of distinctly matched comparison elements CE_m

```
1  method match(CE_p) : CE_m is
2      this.CE_p ← CE_p
3      this.CE_m ← ∅
4      this.O_base ← ∅
5      this.O_comparison ← ∅
6      foreach ce_p ∈ this.CE_p do
7          if ce_p.base ≠ ∅ then
8              this.O_base ← this.O_base ⋃ ce_p.base
9          end
10         if ce_p.comparison ≠ ∅ then
11             this.O_comparison ← this.O_comparison ⋃ ce_p.comparison
12         end
13     end
14     doMatching()
15     return this.CE_m
16 end
```

l. 8: Initialization of the optional base entities.

l. 11: Initialization of the optional comparison entities.

Algorithm 4.5: The main method triggering the Similarity Based Matching (SBM) algorithm.

In Algorithm 4.6, we depict the `doMatching` method. As long as any $ce_p \in CE_p$ still exist, this method checks for the next comparison element, whether it is flagged as *ambiguous* (cf. Line 4) or *handled* (cf. Line 5). The ambiguous flag is used during the processing to mark comparison elements that are in a deadlock with other comparison elements and cannot be matched based on their similarity value sim^{w}_{ce}. In addition, the handled flag is used to prevent infinite processing of comparison elements where the algorithm was not able to find a distinct match in previous executions. By sorting the comparison elements in CE_p whenever a flag is assigned to one of the contained comparison elements $ce_p \in CE_p$, the algorithm always ensures the following order:

1. Comparison elements without flags.
2. Comparison elements with handled flags.
3. Comparison elements with ambiguous flags.

In case the currently processed comparison element ce_p has no assigned flag, the SBM algorithm tries to match it based on a detailed analysis of its relations to all other comparison elements by calling the `tryToMatchCE` method (cf. Line 6). In case the comparison element ce_p was previously marked as handled, the SBM algorithm knows that it already analyzed ce_p and that other comparison elements are preventing the matching. Thus, it first has to resolve any conflicts by calling the `processAmbiguousCEs` method to allow matching of ce_p (cf. Line 8) prior continuing the matching (cf. Line 9). In case the comparison element ce_p was previously marked as ambiguous, the algorithm has to resolve the conflicts by calling the `processAmbiguousCEs` method (cf. Line 12).

```
1  method doMatching() : void is
2      if hasNext(this.CE_p) then
3          ce_p ← next(this.CE_p)
4          if !isAmbiguous(ce_p) then
5              if !isHandled(ce_p) then
6                  tryToMatchCE(ce_p)
7              else
8                  processAmbiguousCEs()
9                  tryToMatchCE(ce_p)
10             end
11         else
12             processAmbiguousCEs()
13         end
14     else
15         createOptionalCEs()
16     end
17     return
18 end
```

l. 1: This method is called until **this**.$CE_p \equiv \emptyset$.

l. 8: Try to match ambiguous elements first.

l. 12: If only ambiguous comparison elements left in **this**.CE_p.

l. 15: If all $ce_p \in$ **this**.CE_p where processed, i.e., **this**.$CE_p \equiv \emptyset$.

Algorithm 4.6: The method controlling the Similarity Based Matching (SBM) algorithm.

In case no $ce_p \in CE_p$ are left (i.e., $CE_p \equiv \emptyset$) the algorithm has to process any model entities that were identified to be optional by calling the `createOptionalCEs` method. Otherwise, these optional comparison elements without an assigned element would not be contained in the list of matched comparison elements CE_m and, thus, not merged in a corresponding 150% model (cf. Section 4.6).

4.5.2. Analyzing the Possible Comparison Elements

In Algorithm 4.7, we depict the `tryToMatchCE` method that analyzes the currently processed comparison element ce_p based on its base entity. Prior to analyzing whether the comparison element ce_p can be matched, the SBM algorithm identifies all other comparison elements CE_p^{sb} with the same base entity (cf. Line 2) and tries to match ce_p based on this information in the remainder of the method. In case the comparison entity contained in ce_p is not set (i.e., the comparison element represents an optional base entity) the SBM algorithm can directly match ce_p by calling the `matchCE` method (cf. Line 4). In case the list of other comparison elements CE_p^{sb} with the same base entity as ce_p is empty, the SBM algorithm continues the matching by analyzing ce_p based on its comparison entities by calling the `matchBasedOnComparisonEntity` method (cf. Line 7). In case the SBM algorithm identifies that other comparison elements $CE_p^b \subset CE_p^{sb}$ exist that have a better similarity value than ce_p, it tries to match one of the identified comparison elements ce_p^b with a higher similarity value instead (cf. Line 11 – 16). In case the SBM algorithm identifies that all identified CE_p^{sb} are in conflict with ce_p, it marks all corresponding comparison elements as ambiguous and sorts them to the end of CE_p (cf. Line 20) and tries to resolve the conflict by continuing the matching of other comparison elements first. In case, none of the above applies (i.e., the comparison

Input: the comparison element ce_p that should be matched

```
1  method tryToMatchCE(ce_p) : void is
2      CE_p^sb ← identifyCEsWithSameBaseEntity(ce_p)
3      if ce_p.comparison ≡ ∅ then
4          matchCE(ce_p)
5          doMatching()
6      else if CE_p^sb ≡ ∅ then
7          matchBasedOnComparisonEntity(ce_p)
8          doMatching()
9      else if betterBaseEntityMatchExists(ce_p, CE_p^sb) then
10         CE_p^b ← identifyBetterCEsInDescOrder(CE_p^sb)
11         foreach ce_p^b ∈ CE_p^b do
12             matchBasedOnComparisonEntity(ce_p^b)
13             if wasCEMatched(ce_p^b) then
14                 break
15             end
16         end
17         doMatching()
18     else if isCEAmbiguous(ce_p, CE_p^sb) then
19         setCEsAmbiguous(ce_p, CE_p^sb)
20         sort(this.CE_p)
21         doMatching()
22     else
23         matchBasedOnComparisonEntity(ce_p)
24         doMatching()
25     end
26     return
27 end
```

l. 2: Identify all CE_p^{sb} with same base entity.

l. 4: Optional base entity identified. Direct match.

l. 7: No other ce_p^{sb} exist. Check whether $ce_p \in$ **this**.CE_p with other comparison entities exist.

l. 10: Identify CE_p^b with better similarity.

ll. 11 – 16: Try to match better ce_p^b.

l. 20: Sort all ambiguous $ce_p \in CE_p$ to the end of **this**.CE_p.

l. 23: Check whether $ce_p \in$ **this**.CE_p with other comparison entities exist.

Algorithm 4.7: The Family Mining match algorithm for base entities of comparison elements.

element ce_p has the highest similarity value compared to comparison elements in CE_p^{sb}), the SBM algorithm continues the matching by analyzing ce_p based on its comparison entities by calling the `matchBasedOnComparisonEntity` method (cf. Line 23).

In Algorithm 4.8, we depict the `matchBasedOnComparisonEntity` method that analyzes the currently processed comparison element ce_p based on its comparison entity. Prior to analyzing whether the comparison element ce_p can be matched, the SBM algorithm identifies all other comparison elements CE_p^{sc} with the same comparison entity (cf. Line 2) and tries to match ce_p based on this information in the remainder of the method. In case the list of other comparison elements CE_p^{sc} with the same comparison entity as ce_p is empty, the SBM algorithm matches ce_p by calling `matchCE` method (cf. Line 4). In case a comparison element with a higher similarity value than ce_p exists in CE_p^{sc}, the SBM algorithm marks ce_p as handled and sorts it in front of the first ambiguous comparison element in CE_p or to the end of CE_p if no ambiguous comparison elements were previously identified (cf. Line 6 – 7). Afterwards, the SBM algorithm identifies the comparison element ce_p^{bsc} with the highest similarity value in CE_p^{sc}, removes its ambiguous flag, that might exist from previous iterations, and moves it to the front of CE_p. This way, we try to match the comparison element ce_p^{bsc} during the next call of the `doMatching` method as it is has a high chance of being matched. In case the SBM algorithm identifies that all identified CE_p^{sc} are in conflict with ce_p, it marks all corresponding comparison elements as ambiguous and sorts them to the end of CE_p (cf. Line 13) and tries to resolve the conflict by continuing the matching of other comparison elements first. In case no better comparison element exists in CE_p^{sc} and the comparison element ce_p is not in conflict with the elements in CE_p^{sc}, the SBM algorithm matches ce_p (cf. Line 15).

4.5.3. Matching the Possible Comparison Elements

When distinctly matching a comparison element ce_p after analyzing its relations to all other CE_p, the `matchCE` method in Algorithm 4.9 is called. It assigns the comparison element to the list of matched comparison elements CE_m and removes it from the list of possible comparison elements CE_p (cf. Line 2 – 3). Afterwards, the algorithm removes the base entity $ce_p.base$ and comparison entity $ce_p.comparison$ in ce_p from the list of optional base entities O_{base} and optional comparison entities $O_{comparison}$, respectively. As matching a comparison element ce_p might resolve existing ambiguities, the SBM algorithm moves corresponding comparison elements to the start of CE_p and continues matching with them. Finally, the handled flag is removed from all remaining comparison elements $ce_p \in CE_p$ to continue matching in clean conditions.

In Algorithm 4.10, we depict the `processAmbiguousCEs` method, which executes the decision wizard as long as any ambiguous comparison elements are left CE_p.

In Algorithm 4.11, we depict the `createOptionalCEs` method, which is called at the end of the SBM algorithm to create optional comparison elements using the specified metric for all entities $e_{base} \in O_{base}$ and $e_{comparison} \in O_{comparison}$, respectively. This way, all entities that were not matched due to ruled out comparison elements are represented by corresponding optional comparison elements ensuring that each model entity is considered in exactly one comparison element.

Input: the comparison element ce_p that should be matched

1 **method** `matchBasedOnComparsionEntity`(ce_p) : **void is**
2 $CE_p^{sc} \leftarrow$ `identifyCEsWithSameComparisonEntity`(ce_p)
3 **if** $CE_p^{sc} \equiv \emptyset$ **then**
4 `matchCE`(ce_p)
5 **else if** `betterCEExists`(ce_p, CE_p^{sc}) **then**
6 `setCEHandled`(ce_p)
7 `sort`(**this**.CE_p)
8 $ce_p^{bsc} \leftarrow$ `identifyBestCE`(CE_p^{sc})
9 `setCEUnambiguous`(ce_p^{bsc})
10 `moveToStartOfCEs`(ce_p^{bsc})
11 **else if** `isCEAmbiguous`(ce_p, CE_p^{sc}) **then**
12 `setCEsAmbiguous`(ce_p, CE_p^{sc})
13 `sort`(**this**.CE_p)
14 **else**
15 `matchCE`(ce_p)
16 **end**
17 **return**
18 **end**

l. 2: Identify all CE_p^{sc} with same comparison entity.

l. 4: No other ce_p^{sc} exist. Direct match.

ll. 6 – 7: Better $ce_p^{sc} \in CE_p^{sc}$ exists. Sort ce_p to back of **this**.CE_p.

ll. 8 – 10: Continue matching with best comparison element.

l. 13: Sort all ambiguous $ce_p \in CE_p$ to the end of **this**.CE_p.

l. 15: No better $ce_p^{sc} \in CE_p^{sc}$ exists and ce_p is *not* ambiguous.

Algorithm 4.8: The Family Mining match algorithm for compare entities of comparison elements.

Input: the comparison element ce_p that was matched

1 **method** `matchCE`(ce_p) : **void is**
2 **this**.$CE_m \leftarrow$ **this**.$CE_m \bigcup ce_p$
3 **this**.$CE_p \leftarrow$ **this**.$CE_p \setminus ce_p$
4 **this**.$O_{base} \leftarrow$ **this**.$O_{base} \setminus ce_p.base$
5 **this**.$O_{comparison} \leftarrow$ **this**.$O_{comparison} \setminus ce_p.comparison$
6 `moveCEsWithResolvedAmbiguounessToStart`(ce_p, **this**.CE_p)
7 `setCEsUnhandled`(**this**.CE_p)
8 **return**
9 **end**

ll. 4 – 5: Remove base and comparison entities from optional entity lists.

l. 6: Continue matching with previously ambiguous elements.

l. 7: Set all $ce_p \in$ **this**.CE_p unhandled.

Algorithm 4.9: The Family Mining algorithm to match a comparison element.

```
1 method processAmbiguousCEs() : void is
2 |  while this.CE_p ≡ ∅ AND ambiguousCEsExist(this.CE_p) do
3 |  |  startDecisionWizard(this.CE_p)
4 |  end
5 |  return
6 end
```

Algorithm 4.10: The Family Mining match algorithm to handle ambiguous comparison elements.

```
1 method createOptionalCEs() : void is
2 |  foreach e_base ∈ this.O_base do
3 |  |  this.CE_m ← this.CE_m ⋃ metricComparison(e_base, ∅)
4 |  end
5 |  foreach e_comparison ∈ this.O_comparison do
6 |  |  this.CE_m ← this.CE_m ⋃ metricComparison(∅, e_comparison)
7 |  end
8 |  return
9 end
```

l. 3: Create new optional *ce* for base entity.

l. 6: Create new optional *ce* for comparison entity.

Algorithm 4.11: The Family Mining match algorithm to handle optional model entities.

4.5.4. Creating Decision Wizards

The decision wizard is an essential part of the SBM algorithm as it allows the language agnostic matching algorithm to solve conflicts that cannot be resolved by matching other comparison elements first. In addition, it allows users to include their domain knowledge by providing resolution strategies for corresponding situations. According to Definition 4.9, we distinguish between manual and automatic resolution strategies.

Manual involvement of domain experts in the algorithms can easily be achieved by providing a GUI to allow presentation of such conflicts and their resolution by selecting one of the shown options. Automatic resolution strategies should only be implemented by developers with sufficient domain knowledge to ensure that the selected comparison elements conform to the assessment of domain experts. Our current strategy for an automatic resolution is to select the comparison element containing base and comparison entities with exactly the same name. If no such comparison element exists, we select the first element from the list of conflicting comparison elements. While this is a naive approach, we identified in our case studies that domain experts approach the problem in a similar manner. Depending on the domain or language, other approaches are applied to find sensible solutions. For example, it would be possible to process all conflicting comparison elements and to analyze the surroundings of the contained entities with a greater radius by not only considering the direct neighbors but also the next one or two successors and predecessors.

To ease creation of decision wizards, our Vamire DSL allows generation of corresponding stubs for an implementation of custom resolution strategies. Furthermore, it allows generation of an implementation of our current name-based algorithm using the `nameBasedDecisionWizard` keyword or the corresponding `EAnnotation` (cf. Table B.1). In case our name-based approach was selected by the developer, the Vampire DSL generator analyzes all meta-model entities inheriting from our base meta-model or using corresponding `EAnnotations` and generates corresponding resolution methods for elements containing a name attribute. For entities that do not have a corresponding attribute, the generator adds annotation warnings for the developer that it was not able to generate corresponding algorithms. Independent of the decision wizard selected for generation, the generated implementations contain a GUI to allow manual selection of resolutions for conflicting comparison elements. Furthermore, we generate a GUI switch for the Family Mining allowing to define prior to each execution whether the developers wants to resolve conflicts manually or using the automatic resolution algorithms.

In Table 4.5, we show all comparison elements that where matched or created using the SBM algorithm based on the input set of possible comparison elements from Table 4.4 for our running example in Figure 2.5. As we can see, only one comparison element was ruled out in stage st_1 as the second comparison element comprises for $mv_{comparison}$ the same transition as the first comparison element and has a higher similarity (i.e., 72%). This matching involved the removal of a transition entity from the list of matched comparison elements CE_m, which was not considered in any other comparison element. Thus, the SBM algorithm created a new optional comparison element comprising this single transition entity.

Due to the language-agnostic realization of our SBM algorithm, it allows to identify distinct matches for possible comparison elements in any modeling language. Similar to the EFA algorithm all language-specific parts (i.e., the creation of optional comparison elements by the metric

and the decision wizards) are realized via called extension points and can be user-adjusted to the current setting. Using our Vampire DSL it is possible to generate large parts of decision wizards for further customization to current settings. Thus, we allow easy adaptation of our SBM algorithm for new languages, while providing interfaces to realize custom-tailored variability mining.

Model Variant	**Contents of Comparison Element** *ce*	**Similarity Value** sim^{w}_{ce}
Stage st_0		
mv_{base}	`cls_unlock`	95%
$mv_{comparison}$	`cls_unlock`	
Stage st_1		
~~mv_{base}~~	~~`key_pos_lock [pw_pos != 1] / cls_locked=true;`~~	~~65%~~
~~$mv_{comparison}$~~	~~`key_pos_lock / cls_locked=true; pw_enabled=false; GEN(pw_but_up);`~~	
mv_{base}	`key_pos_lock [pw_pos == 1] / cls_locked=true; pw_enabled=false;`	72%
$mv_{comparison}$	`key_pos_lock / cls_locked=true; pw_enabled=false; GEN(pw_but_up);`	
mv_{base}	`key_pos_lock [pw_pos != 1] / cls_locked=true;`	0%
$mv_{comparison}$	$\emptyset$	
Stage st_2		
mv_{base}	`cls_lock`	95%
$mv_{comparison}$	`cls_lock`	
Stage st_3		
mv_{base}	`key_pos_unlock / cls_locked=false; pw_enabled=true;`	100%
$mv_{comparison}$	`key_pos_unlock / cls_locked=false; pw_enabled=true;`	

Table 4.5.: Comparison elements matched and created by the Similarity Based Matching (SBM) algorithm for the running example in Figure 2.5, where the annotated stages are highlighted in Figure 4.7.

4.6. Implementing Merging of Variability Information

Based on the comparison elements identified by our SBM algorithm (cf. Section 4.5), it is now possible to generate a 150% model representation of the identified variability that conforms with Definition 2.10. Due to the large diversity of block-based languages (e.g., the different ways to realize hierarchy), it is not possible to realize or generate merging algorithms for such 150% models. Also, developers might have decided to transform syntactic sugar to alternative representations during import, which have to be transformed back to their original representation during export (cf. Section 4.1.3). Otherwise, the generated variability representation as a 150% model might confuse developers and would be useless for them as they might not understand the identified relations. Hence, we regard the 150% model merging as a crucial step for acceptance of the results generated by the Family Mining algorithm and deliberately decided to keep the realization of 150% model merging algorithms as a completely manual step. This way, we give developers full control of the corresponding algorithms and allow them to generate results according to their expectations. In this section, we explain the general ideas of merging the identified variability in a 150% model based on the example of our corresponding implementation for Family Mining of statecharts .

4.6.1. Categorizing the Identified Variability Relations

Before starting the merging of 150% models, developers have to categorize the variability identified by the Family Mining algorithm and stored in the comparison elements.

Definition 4.10: Variability Thresholds

By analyzing the weighted similarity value sim^w_{ce} calculated for compared model entities stored in a comparison element ce (cf. Definition 4.6), it is possible to define a categorization of the identified variability using *variability thresholds*.

To identify fine-grained variability relations following Definition 4.1, we distinguish between *mandatory* comparison elements (i.e., contained entities are regarded as equal despite possible minor differences), *alternative* comparison elements (i.e., contained entities are regarded as mutually-exclusive) and *optional* comparison elements (i.e., contained entities are only part of one of the compared model variants). Following Definition 4.5, these thresholds are defined in the interval $[0..1]$. To select corresponding thresholds, we define the following partial order for the similarity values of comparison elements:

$$sim^w_{ce_{optional}} < sim^w_{ce_{alternative}} < sim^w_{ce_{mandatory}}$$

Based on this definition, developers should select thresholds to categorize the created comparison elements. For the Family Mining of statecharts as well as Matlab/Simulink variants, we use the mapping function $rel(ce)$ in Equation 4.2 to categorize the variability. The used thresholds were selected to allow minor deviations for model elements of mandatory comparison elements (e.g., small changes to the names of states or their interfaces) and proved themselves during work with large-scale models from industry partners.

$$rel(ce) \leftarrow \begin{cases} mandatory & sim^w_{ce} \geq 0.95 \\ alternative & 0 < sim^w_{ce} < 0.95 \\ optional & sim^w_{ce} = 0 \end{cases} \tag{4.2}$$

As the categorization of the identified variability is based on the similarity values calculated by the user-defined metric, we include the definition of such thresholds and the corresponding methods to return the categorization in the specified metrics. In addition, we allow to generate corresponding methods using our Vampire DSL or `EAnnotations` to existing meta-models. In Listing 4.4, we show our extension of the Vampire meta-model description in Listing 4.2 that allows to generate the variability categorization from Equation 4.2 for the metric specified in Listing 4.3. In addition, to allow developers to quickly get started with implementing a custom-tailored merging of 150% models, we generate initial stubs containing annotations with hints on how to merge the variability into a single model.

```
1 // ...
2
3 Thresholds {
4   mandatoryThreshold 0.95
5   optionalThreshold 0.0
6 }
```

Listing 4.4: Vampire DSL definition of variability thresholds used to generate a categorization of the identified variability based on the similarity value sim^{w}_{ce} of comparison elements.

4.6.2. Merging of Statechart 150% Models

To explain the general ideas, challenges and possible solutions for 150% model merging, we present in this section exemplary parts of our merging algorithm for Family Mining of statecharts.

In Algorithm 4.12, we depict the `merge` method that is called by the Family Mining algorithm upon merging all matched comparison elements in a 150% model. The overall merge algorithm relies on a DFS strategy descending into the model hierarchy when identifying any subsequent hierarchy level either realized by sub states or sub regions. We pass the currently compared base model variant $mv_{base} \in MV$ (which can represent a 150% model from a previous iteration of the Family Mining algorithm) and comparison model variant $mv_{comparison} \in MV$ as well as the matched comparison elements for the current iteration of the Family Mining algorithm.

Prior to executing the actual merging, we create a copy of the base model variant mv_{base} that serves as basis for the merged 150% model $m_{150\%}$ and prevents us from altering the actual mv_{base}. During the actual merging, we use a *150% model merge registry* to keep track of different model objects referring to the same model entity when creating the 150% models. Such a registry uses the structure of a map to store the mappings between model objects from different models and their representation in a 150% model. For example, after creating a copy of the base variant for our running example in Figure 2.5a, mv_{base} and the $m_{150\%}$ both contain a model object representing the initial state `cls_unlock`. In addition, when merging model elements from comparison model entities, additional model objects from $mv_{comparison}$ might be added to $m_{150\%}$. Thus, when merging the contents from a comparison element *ce*, we have to look up the corresponding model objects from $m_{150\%}$ or, if not already present (e.g., in case of optional comparison model entities), create and add them. Otherwise, we might not alter the corresponding 150% model objects in $m_{150\%}$ but the original model objects either in mv_{base} or in $mv_{comparison}$.

By initializing such a registry in Line 3 in Algorithm 4.12 with the contents from the freshly initialized 150% model $m_{150\%}$, we enable tracking of all related model objects during the merging.

Input: the compared model variants $mv_{base} \in MV$ and $mv_{comparison} \in MV$ with $mv_{base} \neq mv_{comparison}$ and the matched comparison elements CE_m
Output: the merged 150% model $m_{150\%}$

```
1 method merge(mv_base, mv_comparison, CE_m) : m_150% is
2 |   this.m_150% ← copy(mv_base)
3 |   initializeRegistry(this.m_150%)
4 |   CE^s_m ← identifyStateCEs(CE_m)
5 |   CE^t_m ← identifyTransitionCEs(CE_m)
6 |   mergeRegionContents(m_150%.rootRegion, ∅, CE^s_m, CE^t_m)
7 end
```

l. 2: Basis for the 150% model $m_{150\%}$ is a base model copy.

ll. 4 – 5: Categorize root region sub comparison elements.

Algorithm 4.12: The method triggering the exemplary Family Mining merge algorithm for statecharts.

Next, the algorithm identifies and categorizes the sub state and transition comparison elements from the comparison elements in CE_m containing the compared root regions (cf. Line 4 – 5). Using the 150% model root region $m_{150\%}.rootRegion$ and the categorized state comparison elements CE^s_m and transition comparison elements CE^t_m as initial input, the algorithm starts the merging of the root region contents by calling the `mergeRegionContents` method in Line 6.

General Approach for Merging Model Entities In Algorithm 4.13, we depict the algorithm of the `mergeRegionContents` method. First, it processes the identified state comparison elements CE^s_m and merges them in the hierarchy of the passed region objects (cf. Line 2 – 18). Here r^f_{base} and $r^f_{comparison}$ refer to the corresponding region model objects found in the merge registry. In case one of the state comparison elements ce^s_m contains sub region comparison elements CE^r_m, the algorithm merges them into the previously merged states (cf. Line 8 – 17). Afterwards, all identified transition comparison elements CE^t_m are merged into the hierarchy of the passed region objects (cf. Line 19 – 25). For each merged model element, the algorithm checks whether the variability is merged into a 150% model from a previous merge that already contains variability and selects a corresponding merge strategy. This is an important distinction, because the algorithm has to consider such existing variability (i.e., for the Family Mining iterations $n > 1$) to correctly merge the newly identified variability. For instance, existing alternative groups might need to be extended with new elements or previously mandatory elements have to be changed to optional elements if they are not contained in newly compared variants. In case of the first merging (i.e., for the Family Mining iteration $n = 1$), the algorithm does not need to pay attention to existing variability, but has to initialize correct annotations (e.g., by creating unique identifiers for alternative groups).

The applied ideas for merging regions, states and transitions are basically the same. Thus, we explain them by exemplarily merging transitions into a newly initialized statechart 150% model. As concrete examples, we use the transition comparison elements from Table 4.5 for our running example in Figure 2.5. In Algorithm 4.14, we depict the corresponding algorithm in method `mergeFirstTransitionVariability`. In case that the base model region object was found in the registry, the algorithm merges the currently processed comparison element into the corresponding hierarchy (cf. Line 2 – 29). In case the comparison element does not represent an optional comparison entity (i.e., $t_{base} \neq \emptyset$), the algorithm retrieves the model objects for the base and

Input: the regions r^f_{base} and $r^f_{comparison}$ found in the registry and the comparison elements for states CE^s_m and transitions CE^t_m that should be merged into them

```
1  method mergeRegionContents(r^f_base, r^f_comparison, CE^s_m, CE^t_m) : void is
2      foreach ce^s_m ∈ CE^s_m do
3          if firstVariabilityMerge(this.m_150%) then
4              mergeFirstStateVariability(r^f_base, r^f_comparison, ce^s_m)
5          else
6              mergeFurtherStateVariability(r^f_base, r^f_comparison, ce^s_m)
7          end
8          CE^r_m ← ce^s_m.CE^sub_m
9          if CE^r_m ≠ ∅ then
10             s^f_base ← registryLookup(ce^s_m.base)
11             s^f_comparison ← registryLookup(ce^s_m.comparison)
12             if firstVariabilityMerge(this.m_150%) then
13                 mergeFirstRegionVariability(s^f_base, s^f_comparison, CE^r_m)
14             else
15                 mergeFurtherRegionVariability(s^f_base, s^f_comparison, CE^r_m)
16             end
17         end
18     end
19     foreach ce^t_m ∈ CE^t_m do
20         if firstVariabilityMerge(this.m_150%) then
21             mergeFirstTransitionVariability(r^f_base, r^f_comparison, ce^t_m)
22         else
23             mergeFurtherTransitionVariability(r^f_base, r^f_comparison, ce^t_m)
24         end
25     end
26 end
```

ll. 3, 12 & 20: Check whether variability is merged into an existing 150% model.

ll. 8 – 9: Check for sub region comparison elements in ce^s_m.

ll. 10 – 11: Identify the base and comparison state from ce^s_m in $m_{150\%}$.

Algorithm 4.13: The exemplary Family Mining merge algorithm for statechart region contents.

comparison transition and its source and target states (cf. Line 6 – 13). Otherwise, the algorithm continues merging in Line 27 to add the optional comparison transition to the 150% model. Next, the algorithm uses the threshold-based variability categorization introduced in Section 4.6.1 to distinguish between mandatory comparison elements (cf. Line 15), alternative comparison elements (cf. Line 17) and optional comparison elements (cf. Line 23).

In case a mandatory transition has to be merged (e.g., for the transition comparison element in stage st_3 of our running example in Table 4.5), the algorithm does not add any new model object to the 150% model. Instead, it calls the `setTransitionMandatory` method to annotate the existing base transition object t^f_{base} from the 150% model with information about the containing models

Input: the regions r^f_{base} and $r^f_{comparison}$ found in the registry and
the transition comparison element ce^t_m that should be merged into them

```
1  method mergeFirstTransitionVariability(r^f_base, r^f_comparison, ce^t_m) : void is
2      if r^f_base ≠ ∅ then
3          t_base ← ce^t_m.base
4          t_comparison ← ce^t_m.comparison
5          if t_base ≠ ∅ then
6              t^f_base ← registryLookup(ce^t_m.base)
7              s^fs_base ← registryLookup(t_base.source)
8              s^ft_base ← registryLookup(t_base.target)
9              s^fs_comparison ← ∅
10             s^ft_comparison ← ∅
11             if t_comparison ≠ ∅ then
12                 s^fs_comparison ← registryLookup(t_comparison.source)
13                 s^ft_comparison ← registryLookup(t_comparison.target)
14             end
15             if isMandatory(ce^t_m) then
16                 setTransitionMandatory(t^f_base, t_comparison)
17             else if isAlternative(ce^t_m) then
18                 if sourceAndTargetAreMandatory(s^fs_base, s^ft_base, s^fs_comparison, s^ft_comparison) then
19                     addAlternativeTransitionGroup(s^fs_base, s^ft_base, ce^t_m)
20                 else
21                     ...
22                 end
23             else if isOptional(ce^t_m) then
24                 setTransitionOptional(t^f_base)
25             end
26         else
27             mergeOptionalComparisonTransitionsFirstVariability(r^f_comparison, ce^t_m)
28         end
29     else
30         mergeOptionalComparisonTransitionsFirstVariability(r^f_comparison, ce^t_m)
31     end
32 end
```

l. 2: Check, whether to merge into the base region.

l. 5: *Not* an optional comparison transition.

l. 6 – 8: Identify the base transition and its source and target states in $m_{150\%}$.

l. 12 – 13: Identify the comparison transition's source and target states in $m_{150\%}$.

ll. 27 & 30: Merge an optional comparison transition to the 150% model.

Algorithm 4.14: The exemplary Family Mining merge algorithm for statechart transitions.

(i.e., mv_{base} and $mv_{comparison}$), and its variability (i.e., mandatory). Furthermore, the algorithm adds details about any differing attributes found in $t_{comparison}$ (e.g., a renamed event triggering the transition) that can exist in cases where the mandatory threshold is below 100% allowing for minor deviations. Without this additional annotation, the merging algorithm would loose information as these deviations are not documented in the 150% model. Similarly, the `setTransitionOptional` method annotates for merged optional transitions (e.g., for the transition comparison element with 0% similarity in stage st_1 of our running example in Table 4.5) that the corresponding transition object represents an optional model entity and in which models it is contained (i.e., mv_{base}).

In contrast, the merging of alternative comparison elements involves more logic as different scenarios have to be considered. In our exemplary excerpt from the `mergeFirstTransitionVariability`, we only depict the merging of alternative transitions where the transitions' source and target states were previously identified to be mandatory (e.g., for the transition comparison element with 72% similarity in stage st_1 of our running example in Table 4.5). In this case, we have to add a copy of the comparison transition object $t_{comparison}$ to the 150% model and set its source and target states to s^{fs}_{base} and s^{ft}_{base}, respectively. Furthermore, we have to add the copied transition object as outgoing transition for the source state and as incoming transition for the target state to allow traversal of the newly added alternative path in the 150% model. In addition, we annotate for the alternative transitions that they form an alternative group (i.e., by marking them as alternative and adding a unique group id) and from which models the transition objects originate (i.e., mv_{base} and $mv_{comparison}$, respectively). Other possible scenarios for merging alternative comparison elements (not shown in Algorithm 4.14 due to space limitations) are cases where the source and target states were identified as alternatives or only one of the states represents an alternative. During the merging of these cases, the algorithm similarly has to copy the alternative transition from $mv_{comparison}$ and correctly set all necessary relations between the objects to allow traversal of the 150% model. This is essential to allow our EFA algorithm to automatically traverse 150% models for iterations $n > 1$ of the algorithm to consider all alternative paths during comparison of further models.

Merging of Container Entities During our work with large-scale models, we realized that in certain situations the merging of container entities results in unsatisfactory 150% models. In Figure 4.9, we present a simplified example comprising a number of regions that should be merged into a 150% model. The Family Mining would correctly identify that `Region A` is mandatory (shown by the exclamation mark), `Region B1` and `Region B2` are alternative (shown by the double-headed arrow), because state `B2` was replaced by `B3`, and `Region E` is optional (shown by the question mark). However, due to the minor similarity of `Region C` and `Region D` (because the transitions' triggering events are equal) the algorithm would identify these regions as alternatives, which might not be desired as they possibly realize unrelated features. To overcome this problem, two solutions can be applied. First, the threshold for optional comparison elements can be raised in order to, for example, identify elements with a similarity of up to 40% as separate optional elements. Second, it is possible to compare the names of the containers using the Levenshtein Distance algorithm [Lev66] and to regard containers with a name similarity below a user-adjustable threshold (we used 80% as default value) as separate optional elements. In both situations, the merge algorithm has to implement additional logic to execute a "splitting" of comparison elements comprising two separate optional containers. Depending on the current setting either solution is appropriate

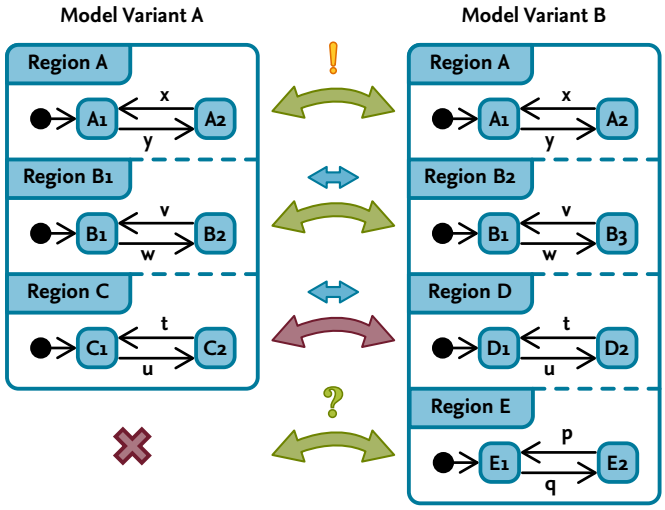


Figure 4.9.: Simplified example for a problematic merging of regions into the 150% model.

Input Variant Model Object		150% Model Object
`ManPW:S:cls_unlock`	→	`S:cls_unlock`
`AutoPW:S:cls_unlock`	→	`S:cls_unlock`
`ManPW:S:cls_lock`	→	`S:cls_lock`
`AutoPW:S:cls_lock`	→	`S:cls_lock`
`ManPW:T:key_pos_lock:pw_pos_neq_1`	→	`T:key_pos_lock:pw_pos_neq_1`
`ManPW:T:key_pos_lock:pw_pos_eq_1`	→	`T:key_pos_lock:pw_pos_eq_1`
`AutoPW:T:key_pos_lock:gen_pw_but_up`	→	`T:key_pos_lock:gen_pw_but_up`
`ManPW:T:key_pos_unlock`	→	`T:key_pos_unlock`
`AutoPW:T:key_pos_unlock`	→	`T:key_pos_unlock`

Table 4.6.: The exemplary 150% model merge registry contents after merging the matched comparison elements from Table 4.5 for our running example in Figure 2.5.

to solve such situations. For our current implementation of Family Mining for statecharts as well as Matlab/Simulink models, we apply splitting based on names and were able to generate results conforming to the expectations of domain experts. While, in theory, similar problems can occur for nodes or edges, we did not encounter them during our work with academic and industrial models. Thus, we did not implement similar algorithms for these elements. However, during a custom implementation of the merging algorithms such a solution can easily be added if necessary.

To allow merging of comparison elements in the 150% model, we have to update the merge registry during the whole merging process. In Table 4.6, we show the corresponding contents of a 150% model merge registry after merging the matched comparison elements from Table 4.5 for our running example in Figure 2.5. The shown identifiers were selected to be human-readable, while, depending on the used mapping for the 150% model merge registry, the mapped objects would normally be referenced based on corresponding hash values. On the left side of the ta-

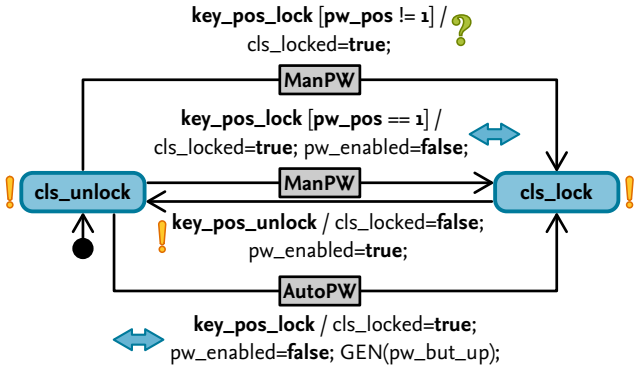


Figure 4.10.: The 150% model merged for the matched comparison elements from Table 4.5 for the Central Locking System (CLS) variants from our running example in Figure 2.5.

ble, we show the state objects (denoted with `S`) and transition objects (denoted with `T`) from the ManPW and AutoPW model variants of the CLS systems. The remaining identifier after the model variants and the object type abstracts the corresponding model objects with an identifier that is as short as possible, but expressive enough to make the objects distinguishable. For example, `ManPW:T:key_pos_lock:pw_pos_neq_1` abstracts from transition `key_pos_lock [pw_pos != 1] / cls_locked=true;`. On the right side of the table, we show the mapping to corresponding 150% model objects. For example, in case of the `cls_unlock` states from both variants, we can see that they were identified as mandatory. As a result, both model objects are represented and mapped to the same object in the 150% model.

In Figure 4.10, we show a visual representation of the 150% model created by our merge algorithm for Family Mining of statecharts from the matched comparison elements in Table 4.5 for our running example in Figure 2.5. As we can see, the merge algorithm correctly identified and annotated both states and the transition from state `cls_lock` to `cls_unlock` as mandatory (shown by the exclamation mark). Furthermore, the alternative relation between the two transitions containing the `pw_enabled=false` action are marked as alternative (shown by the double-headed arrow) and the remaining transition as optional (shown by the question mark). The boxes containing a model name show which of the model variants contain the corresponding entities. We do not visualize these annotations for mandatory entities as they are contained in all compared variants and to improve the readability of the visualized 150% model.

Using custom implementations for the merging of 150% models from the identified variability relations stored in corresponding comparison elements, developers are now able to generate a single representation of the variability comprised in the analyzed model family. Based on this representation, detailed analyses of relations between the model variants are possible and adjustments to the identified relations can be executed (e.g., by changing the identified explicit variability or adding new variants). In addition, the overall maintenance of the analyzed variants is eased. For instance, when identifying a bug in a specific variant, it is now possible to identify in the 150% model which other variants contain the same implementation details. Based on such information, it is possible to directly focus on the corresponding variants without executing tedious manual comparisons between all variants prior to fixing the problem. Furthermore, a migration of the variants to managed reuse in an SPL can be achieved (cf. Chapter 5).

4.7. Identifying Hierarchy Shifts and Horizontal Dispersions of Model Parts

While the Family Mining algorithms described in the previous sections already identify reliable variability relations between model variants for most cases, we identified situations where the algorithms might not identify the expected variability relations. An example for such a situation is our running example in Figure 2.6. Here, the developer moved the implementation in the hierarchy. Such editing steps introduce hierarchy shifts.

Definition 4.11: Hierarchy Shifts

Hierarchy shifts move developed functionality in form of encapsulated sub models from one hierarchy level to another.

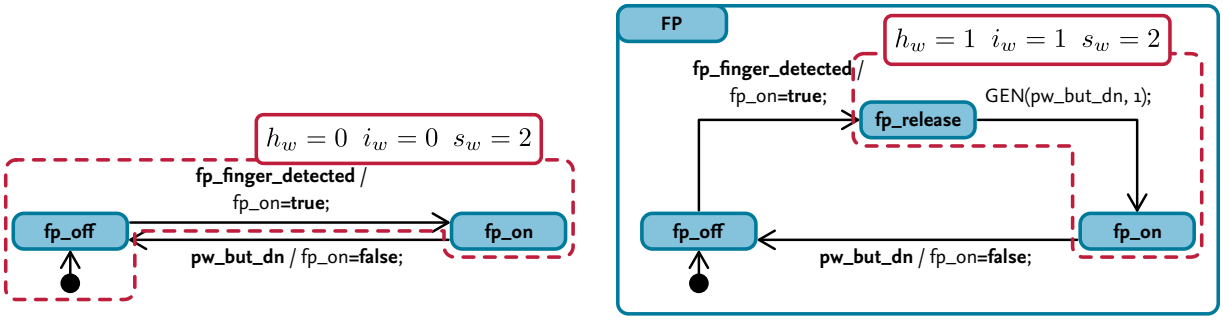
For instance, in the development environment of Matlab/Simulink, it is possible to select parts of a hierarchy level and encapsulate them in a newly created subsystem. Furthermore, in the example in Figure 2.6, the developer changed the behavior of the implementation by adding the `fp_release` state and rerouting the transitions. Such editing steps introduce horizontal dispersions.

Definition 4.12: Horizontal Dispersions

Horizontal dispersions introduce new node entities on a path between existing node entities (e.g., to add bug fixes) and, thus, introduce new behavior on these paths.

For instance, in Matlab/Simulink additional *Gain Blocks* are sometimes introduced to amplify signals. Due to the nature of our EFA algorithm, our Family Mining would not correctly identify the variability relations in such scenarios. On the one hand, it would not identify the relation between the two variants of the FP in Figure 2.6 as they are not on the same hierarchy level. Instead, the algorithm would match the model elements from the variant in Figure 2.6a to other elements at the same hierarchy level of hierarchical state `FP` in Figure 2.6b (not shown to simplify the example). On the other hand, the EFA algorithm is not capable of identifying insertions, such as the `fp_release` state, because no connection between the `fp_off` and `fp_on` states exist in Figure 2.6b. Thus, following the execution flow in both models, the algorithm would identify a relation between the two `fp_off` states and the `fp_on` and `fp_release` states. In addition, the `fp_on` state in Figure 2.6b would be regarded as optional as no matching partner exists in Figure 2.6a. For simplicity, we did not discuss the implications for transitions in this example. However, similar issues would apply.

During work with our industry partner, we identified that such scenarios are rarely occurring in case of their company. In interviews concerning the applicability of our Family Mining approach, the developers stated that the main reason is that such edits are conflicting with the companies development guidelines (for a detailed report on the interviews, we refer to Section 7.5) and are only applied in exceptional situations (e.g., to fix an error prior to a major release that cannot be delayed). Despite the identified rarity of such scenarios, we came to agree that solutions for such scenarios increase the generalizability of our Family Mining algorithm.



(a) The unchanged variant of the FP system with an exemplary matching window.

(b) The edited variant of the FP system with an exemplary matching window.

Figure 4.11.: Exemplary matching windows added to the Finger Protection (FP) variants in Figure 2.6a.

4.7.1. Generating Window Pairs using the Matching Window Analysis Algorithm

To be able to identify such horizontal dispersions and hierarchy shifts, we developed the Matching Window Analysis (MWA) algorithm. The general idea is to virtually divide the analyzed models into smaller sections that are also compared across hierarchies and to generate additional comparison elements using the EFA algorithm (cf. Section 4.4) inside these sections.

Definition 4.13: Diameter

In accordance to Definition 4.7, the *diameter* d defines the maximum number of node entity stages contained in the currently considered model hierarchy.

For the two models in Figure 2.6, the diameter is $d_{(a)} = 2$ and $d_{(b)} = 3$, respectively. Using this definition, we can now define the considered sections of models and their corresponding parameters. We call these sections matching windows.

Definition 4.14: Matching Windows

A *matching window* (or for short *window*) w divides a hierarchy level (e.g., a Matlab/Simulink subsystem) of a model into a smaller scope and, thus, does not consider the complete hierarchy level for comparison. Such windows are defined by the:

- *diameter* d of the current hierarchy level (cf. Definition 4.13).

and their:

- *starting position* i_w in relation to node entity stage st_0 in the interval $0 \leq i_w < d$.
- *size* s_w defining contained node entity stages st in the interval $s_{min} \leq s_w < (d - i_w)$.
- *hierarchy level* h_w of the window in the interval $0 \leq h_w \leq h_{max}$.

In Figure 4.11, we indicate two exemplary windows with corresponding parameters. Both windows have the same size $s_w = 2$, but different starting positions $i_w = 0$ and $i_w = 1$, respectively. In addition, the window in Figure 4.11b is on a different hierarchy level $h_w = 1$. We do not include the

stages for edge entities in the diameter calculation as, otherwise, we would allow dangling edges in the created windows. For example, in Figure 4.11b, we would allow to have a window of size $s_w = 4$ (assuming we are including edges) that contains the `fp_off` state, the `fp_release` state, the transition between these states and also the transition going to state `fp_on`, but not the target state `fp_on` itself. As a result, additional logic would be necessary to properly align windows, such as, to check that a window starting with a stage of node entities is not compared with a window starting with a stage of edges. Thus, the diameter and the corresponding windows assume that we always create windows between node entities. However, the edge entities on the corresponding paths are still considered during comparisons of the windows by the EFA algorithm.

To allow developers to adjust the considered hierarchy levels and windows sizes, we allow to adjust the maximum hierarchy h_{max} to be analyzed (where $h_{max} = 0$ disables the search for hierarchy shifts) and the minimum and maximum window size s_{min} and s_{max}, respectively. By default, we automatically initialize s_{max} with $\texttt{min}(d_{base}, d_{comparison})$ identifying the smallest diameter of the compared hierarchy levels from both models (i.e., $s_{max} = 2$ for the example in Figure 4.11). Based on this, s_{min} is initialized with $\lceil \frac{s_{max}}{fraction} \rceil$, where *fraction* controls the fraction of stages that should be analyzed and we ensure that $2 \leq s_{min} \leq \lceil \frac{s_{max}}{fraction} \rceil$ holds. Otherwise, using $s_{min} = 1$ would basically emulate the EFA algorithm as single stages would be created and compared. By default the fraction parameter is initialized to $fraction = 2$. Thus, we use $s_{min} = 2$ for the example in Figure 4.11.

Definition 4.15: Window Pair

Using two windows w_{base} and $w_{comparison}$ created in accordance with Definition 4.14, we can create a *window pair* wp that defines two model sections from the base and comparison model, respectively. These sections are then compared using the EFA algorithm (cf. Section 4.4).

In Algorithm 4.15, we depict the algorithm of the `analyzeModels` method used to walk through all hierarchy levels of the compared models and generate and compare window pairs. First, the algorithm identifies all sub models in the model hierarchy of compared models (cf. Line 4 – 5). Next, the algorithm generates all possible window pairs for $0 \leq h_c \leq h_{max}$ (cf. Line 6 – 15). As the MWA algorithm is realized as an alternative to the EFA algorithm (while using the EFA algorithm internally to compare windows), it is also called recursively for each found hierarchical model element. Thus, the algorithm considers all hierarchy levels, but is able to search for hierarchy shifts for each hierarchy level up to h_{max} levels. After generating all possible window pairs WP for an iteration, the algorithm identifies the best window pair wp_b from WP based on the average similarity of the contained comparison elements (cf. Line 16) and executes a post processing for wp_b (cf. Line 17). Due to the nature of the MWA algorithm, this step is necessary to include all model elements from the compared hierarchy levels that were not considered in wp_b (i.e., model elements surrounding the created window pair).

In Algorithm 4.16, we depict the algorithm of the `generateWindowPairs` method used to generate the window pairs for two given sub models on potentially different hierarchy level. The *creationLoop* in Line 10 – 30 is iterated until all window pairs with $s_{min} \leq s \leq s_{max}$ are created by decrementing the window size for each iteration (cf. Line 29). For each iteration, the algorithm iterates over all possible start positions i_w for the compared base and comparison model and creates corresponding window pairs of size s_w (cf. Line 11 – 12). In case the similarity sim^w_{wp} of the cre-

Input: $mv_{base} \in MV$ and $mv_{comparison} \in MV$ with $mv_{base} \neq mv_{comparison}$ and the selected maximum hierarchy h_{max}
Output: a list of distinctly matched comparison elements CE_m

```
1  method analyzeModels(mv_base, mv_comparison, h_max) : CE_m is
2      WP ← ∅
3      MV_h ← ∅
4      MV_h ← MV_h ⋃ identifySubModels(mv_base)
5      MV_h ← MV_h ⋃ identifySubModels(mv_comparison)
6      for h_c ← 0; h_c ≤ h_max; h_c ← h_c + 1 do
7          MV_h_c ← identifyCurrentSubModels(MV_h, h_c)
8          for mv_h ∈ MV_h_c do
9              if mv_h ⊂ mv_base then
10                 WP ← WP ⋃ generateWindowPairs(mv_h, mv_comparison, h_c, 0)
11             else if mv_h ⊂ mv_comparison then
12                 WP ← WP ⋃ generateWindowPairs(mv_base, mv_h, 0, h_c)
13             end
14         end
15     end
16     wp_b ← findBestWindowPair(WP)
17     postProcessWindowPair(wp_b)
18     return wp_b.CE_m
19 end
```

ll. 4 & 5: Identify hierarchy sub models MV_h.

l. 17: Create and include comparison elements for unconsidered model elements outside wp_b.

Algorithm 4.15: The main method triggering the Matching Window Analysis (MWA)

ated window pair wp exceeds the threshold for mandatory comparison elements, we assume that it represents a direct match and break the window pair creation. Using this heuristic, we reduce the number of unnecessarily created window pairs as they most likely will have a lower similarity. After creating all possible window pairs and not identifying a direct match, we try to identify a horizontal dispersion window pair wp_{hd} (cf. Line 21) and directly match this dispersion (cf. Line 23). Details on the dispersion identification can be found in Section 4.7.2. Otherwise, the algorithm stores all window pairs identified for the current s_w (cf. Line 26).

Input: models $mv_{base} \in MV$ and $mv_{comparison} \in MV$ with $mv_{base} \neq mv_{comparison}$, where one of the models can be a sub hierarchy, together with hierarchy levels h_{base} and $h_{comparison}$ and s_{min} and s_{max} controlling the size of the created windows
Output: all identified window pairs WP

```
1  method generateWindowPairs(mv_base, mv_comparison, h_base, h_comparison) : WP is
2      WP_tmp ← ∅
3      WP ← ∅
4      threshold ← getMandatoryThreshold()
5      d_base ← calculateDiameter(mv_base)
6      d_comparison ← calculateDiameter(mv_comparison)
7      s_max ← min(d_base, d_comparison)
8      s_min ← ⌈s_max / fraction⌉
9      s_w ← s_max
10     creationLoop : while s_w ≥ s_min do
11         for i_base ← 0; i_n ≤ d_base − s_w; i_base ← i_base + 1 do
12             for i_comparison ← 0; i_comparison ≤ d_comparison − s_w; i_comparison ← i_comparison + 1 do
13                 wp ← createWindowPair(i_base, h_base, i_comparison, h_comparison, s_w)
14                 WP_tmp ← WP_tmp ⋃ wp
15                 if sim^w_wp ≥ threshold then
16                     WP ← WP ⋃ wp
17                     break creationLoop
18                 end
19             end
20         end
21         wp_hd ← getHorizontalDispersionWindowPair(WP_tmp)
22         if wp_hd ≠ ∅ then
23             WP ← WP ⋃ wp_hd
24             break creationLoop
25         else
26             WP ← WP ⋃ WP_tmp
27         end
28         WP_tmp ← ∅
29         s_w ← s_w − 1
30     end
31     return WP
32 end
```

l. 10: Create all window pairs with $s_w \geq s_{min}$.

ll. 11 & 12: Iterate over possible start positions i_w and create window pairs.

l. 16: Directly match in case threshold is exceeded.

l. 23: Directly match in case a horizontal dispersion was identified.

Algorithm 4.16: The method generating window pairs for the Matching Window Analysis (MWA)

4.7.2. Identifying Horizontal Dispersions

To identify a dispersion, we generate the window pairs and corresponding comparison elements by executing the `analyzeModels` method in Algorithm 4.15. Next, we apply a trick to identify the distinct variability relations, which is possible due to the nature of our SBM algorithm. We retrieve all matched comparison elements identified for the window pairs in WP_{tmp} and apply the SBM algorithm a second time. This way, we are able to remove potentially duplicate comparison elements (created for different window pairs) and identify best matches across the windows.

In Table 4.7, we show exemplary window pairs with comparison elements that allow to identify the horizontal dispersion in Figure 2.6. For space reasons, we only included the window pairs wp_0, wp_1 and wp_5 relevant for the detection in this example. The comparison elements from the remaining window pairs would be removed by the matching algorithm due to lower similarity values. As we can see, the algorithm is able to correctly identify that the `fp_release` state and the `GEN(pw_but_dn, 1)` transition represent optional elements. This is possible as the algorithm executes the matching across window pair borders and, thus, can identify better matches for the `fp_on` state and the `fp_finger_detected / fp_on=true;` transition that, otherwise, would not be possible. In addition, duplicate comparison elements between wp_0 and wp_5 are removed.

The `getHorizontalDispersionWindowPair` method called in Algorithm 4.16 can identify the horizontal dispersions using the following definition.

Definition 4.16: Extension of Definition 4.12 for Horizontal Dispersions

A *horizontal dispersion* can be identified if an *optional path* between a mandatory / alternative model element and another mandatory / alternative model element exists. This optional path can consist of an arbitrary number of optional node and edge entities.

Based on this definition, the algorithm is able to detect optional elements that only exist in specific variants of the family and which have a path starting and ending at model elements which are mandatory or alternative. This way, the algorithm can assume that the start and end elements are contained in all variants (i.e., they are mandatory or part of an alternative group that comprises mutually exclusive parts for all variants). As a result, the optional path between the start and end elements can be regarded as an insertion as it is only present in certain variants. For our example, the `getHorizontalDispersionWindowPair` method, thus, identifies that an optional path (i.e., a horizontal dispersion) exists between the alternative `fp_finger_detected / fp_on=true;` transition present in the FP variant with a hierarchy shift (cf. Figure 2.6b) and the mandatory `fp_on` state. After identifying the horizontal dispersion the corresponding comparison elements are included in a new window pair.

The 150% model resulting from a merging of the identified comparison elements containing the hierarchy shift and the horizontal dispersion can be found in Figure 4.12. During the creation of such 150% models, we have to additionally annotate the possible hierarchy levels of the merged model elements. For instance, hierarchical state `FP` and transition `fp_finger_detected / fp_on=true;` between the states `fp_off` and `fp_on` are only present on hierarchy level 0, while alternative `fp_finger_detected / fp_on=true;` and all optional parts are only present on hierarchy level 1. The hierarchy level of all mandatory parts depends on the variant (i.e., hierarchy level 0 for the `FP` variant without and hierarchy level 1 for the `FP_h` variant with a hierarchy shift).

Model Variant	Contents of Comparison Element ce	Similarity Value sim_{ce}^{w}
Window Pair wp_0: $w_{base}(h_w = 0, i_w = 0, s_w = 2)$ – $w_{comparison}(h_w = 1, i_w = 0, s_w = 2)$		
mv_{base}	`fp_off`	100%
$mv_{comparison}$	`fp_off`	
mv_{base}	`fp_finger_detected / fp_on=true;`	85%
$mv_{comparison}$	`fp_finger_detected / fp_on=true;`	
mv_{base}	~~`fp_on`~~	~~80%~~
$mv_{comparison}$	~~`fp_release`~~	
Window Pair wp_1: $w_{base}(h_w = 0, i_w = 0, s_w = 2)$ – $w_{comparison}(h_w = 1, i_w = 1, s_w = 2)$		
mv_{base}	~~`fp_off`~~	~~50%~~
$mv_{comparison}$	~~`fp_release`~~	
mv_{base}	$\emptyset$	0%
$mv_{comparison}$	`fp_release`	
mv_{base}	~~`fp_finger_detected / fp_on=true;`~~	~~0%~~
$mv_{comparison}$	~~`GEN(pw_but_dn, 1)`~~	
mv_{base}	$\emptyset$	0%
$mv_{comparison}$	`GEN(pw_but_dn, 1)`	
mv_{base}	`fp_on`	100%
$mv_{comparison}$	`fp_on`	
Window Pair wp_5: $w_{base}(h_w = 0, i_w = 1, s_w = 2)$ – $w_{comparison}(h_w = 1, i_w = 2, s_w = 2)$		
mv_{base}	~~`fp_off`~~	~~100%~~
$mv_{comparison}$	~~`fp_off`~~	
mv_{base}	`pw_but_dn / fp_on=false;`	100%
$mv_{comparison}$	`pw_but_dn / fp_on=false;`	
mv_{base}	~~`fp_on`~~	~~100%~~
$mv_{comparison}$	~~`fp_on`~~	

Table 4.7.: Exemplary window pairs wp_0, wp_1 and wp_5 with corresponding comparison elements generated for the running example in Figure 2.6 to illustrate the detection of horizontal dispersions.

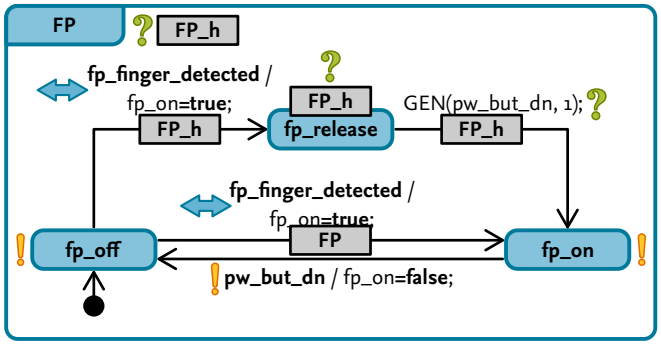


Figure 4.12.: The 150% model merged for the matched comparison elements from Table 4.7 for the Finger Protection (FP) variants from our running example in Figure 2.6.

Overall, the presented MWA algorithm allows to improve the variability identification in cases where hierarchy shifts and horizontal dispersions are introduced during clone-and-own of variants. The information about such editing can provide developers with valuable insights how to align the clone-and-own variants (e.g., by applying the hierarchy shift in all variants). For instance, such an alignment can improve results during a migration to an SPL (cf. Chapter 5).

4.8. Related Work

The work related with our Family Mining approach consists of techniques to identify fine-grained variability information for different related development artifacts. The analyzed artifacts can be categorized into *source code artifacts* (i.e., textual artifacts as Java or C++ code) and *modeling artifacts* (e.g., statechart models or UML models).

In general, approaches to identify variability can be categorized into *clone detection* (cf. Section 4.8.1), *differencing* (cf. Section 4.8.2) and *hybrid techniques* (cf. Section 4.8.3), which combine information on clones and differences. Similar to hybrid techniques, the discussed differencing techniques also rely on information of cloned parts to identify the differences between compared artifacts. However, to make a clear distinction between algorithms whose primary goal is the identification of such *differences* (e.g., to present them to the user) and approaches whose goal is to use the commonalities and differences to *merge a shared model* (e.g., in case of versioning or creation of 150% models in the context of SPLs), we introduce hybrid techniques as an additional category. This allows clear separation and permits us to put emphasis on *hybrid techniques* that are most related to our approach.

While we acknowledge and discuss approaches targeting identification of variability information in source code, we put our emphasize on the variability identification for models as these algorithms are most related to our Family Mining approach.

4.8.1. Clone Detection Techniques

Clones of software fragments or complete implementation artifacts occur during the development of software for different reasons [BKA+07, Kos07, RC07, RCK09]. For instance, they provide an easy means to reuse existing functionality and might even reduce the risk of breaking existing code in settings with high reliability requirements (i.e., copying well tested code might be more secure than developing potential problematic new code). Code duplication might also occur in settings where the used language does not provide suitable techniques for reuse (e.g., inheritance or generics).

Furthermore, developers might reimplement existing solutions in large companies (e.g., with plants across different countries) as they might not be aware of them.

In literature, many definitions and categorizations of different clone types exist, which boil down to the following four types [BKA+07, Kos07, RC07, RCK09]:

- *Type 1 Clones:* These are completely equivalent clones with changes only occurring in used comments or whitespace (i.e., the layout of the code).
- *Type 2 Clones:* In addition to the differences of *Type 1 Clones*, these clones allow changes to identifiers, literals or used types.
- *Type 3 or Near-Miss Clones:* In addition to the differences of *Type 2 Clones*, these clones allow changes to statements, such as additions, deletions or modifications.
- *Type 4 or Semantic Clones:* This category comprises clones that are semantically equivalent (i.e., compute the same results), but are implemented using different syntactical solutions (e.g., calculating 2×2 vs. 2^2).

While using clones is often regarded as a negative programming strategy (e.g., erroneous code might be duplicated and has to be manually identified in all artifacts to fix corresponding bugs), it can also have positive effects when applying it with caution and in a systematic way [KG06]. For example, developers can test optimizations of the existing code without breaking core functionality of the developed system.

Source Code Clone Detection However, due to the mainly negative connotation of clones, efficient and accurate identification of software clones has been a research topic for many years and, thus, a variety of corresponding approaches exists for either source code [BKA+07, Kos07, RC07, RCK09] or models [SC12, SC13]. For our discussion of source code clone detection algorithms, we use the categorization from [RC07]:

Text-Based Source Code Clone Detection Examples for *text-based source code clone detection* are the algorithms by Johnson [Joh93, Joh94], Ducasse et al. [DRD99] and Roy et al. [RC08, Roy09, CR11]. For instance, Roy et al. [RC08, Roy09, CR11] describe an approach that is based on *Turing Extender Language (TXL)* grammars to parse, normalize and compare source code artifacts. The TXL was specifically designed to allow rapid prototyping of languages by applying source transformations to an original TXL-based language and allows rule based transformations of code [Cor06]. To identify cloned source code, the authors use TXL grammars to parse the input artifacts and identify potential clones based on TXL descriptions of minimal clones (i.e., the smallest entity that should be considered for clone detection). By applying flexible pretty-printing, the authors transform the parsed code in a uniform way to enable their comparison and further tailor the compared artifacts to the correct granularity (e.g., it is possible to compare complete loop expressions at once or break them into separate lines). To compare the pretty-printed code, the authors apply user-adjustable code normalization allowing to identify near-miss code clones (e.g., renamed variables) and calculate clones based on longest common subsequences and the ratio of unique items. The identified clone clusters afterwards can be analyzed based on a textual output or a generated *Hypertext Markup Language (HTML)* report.

Token-Based Source Code Clone Detection Examples for *token-based source code clone detection* are the algorithms by Baker [Bak92, Bak95], Kamiya et al. [KKI02] and Li et al. [LLM+04, LLM+06]. For instance, Kamiya et al. [KKI02] apply language-specific transformation rules to a token stream created by a lexical parser for the analyzed language. Their tool Code Clone Finder (CCFinder) uses transformation rules to execute parameter replacement after applying language specific transformations (e.g., removing namespace attributes, such as `std::`). Based on the code with replaced parameters the authors identify sequences of matching lines. The identified clones and near-miss clones are reported to the user by transforming the found passages back to their original representation.

Metric-Based Source Code Clone Detection Examples for *metric-based source code clone detection* are the algorithms by Mayrand et al. [MLM96] and Di Lucca et al. [DDF02]. For instance, Di Lucca et al. compare HTML or *Active Server Page (ASP)* documents based on their edit distances. One described approach translates the tags in HTML documents and the built-in elements in ASPs to strings where they are replaced by an element from a corresponding simplified alphabet (e.g., a `<td>` tag is replaced by an `a`) [DDF02]. The resulting strings are compared using the distance calculated by the Levenshtein Distance algorithm [Lev66]. A second approach uses the frequency of tags occurring in HTML documents to compare them. The resulting distances or frequencies allow the developers to identify exact copies or near-miss clones and, in fact, where used to identify plagiarism.

Tree-Based Source Code Clone Detection Examples for *tree-based source code clone detection* are the algorithms by Baxter et al. [BYM+98] and Wahler et al. [WSW+04]. For instance, Baxter et al. [BYM+98] translate the analyzed source code to ASTs using corresponding parsers and detect cloned parts based on this representation. By applying a hash function, the authors compare sub trees of the generated AST and use a similarity function considering the equivalent and differing nodes of compared sub trees to allow near-miss clones above a certain threshold to be added to the same hash buckets. Using a list structure for sequences of the programs and connecting them with the hash codes of corresponding sub trees, the algorithm is capable of identifying sequences of cloned parts.

Program-Dependence-Graph-Based Source Code Clone Detection In general *Program Dependence Graphs (PDGs)* are graph representations of the analyzed code showing the control and data dependencies of programs [FOW87]. Thus, they provide detailed information that can be exploited for clone detection. Examples for *program-dependence-graph-based source code clone detection* are the algorithms by Krinke [Kri01] and Komondoor et al. [KH01]. For instance, Krinke [Kri01] describes how PDGs enriched with additional information (e.g., attributed vertices) can be used to identify clones. Beginning at the PDG nodes representing a dependence start the author calculates maximal similar subgraphs by identifying subsequent vertices based on outgoing edges and assigned attributes. Due to the used PDG structure, algorithms in this category are much more aware of the semantics in programs and do not solely rely on syntactical information.

In addition, there also exist hybrid clone detection techniques using combinations of the presented approaches from the previous paragraphs (e.g., Koschke et al. [KFF06] use suffix trees in conjunction with ASTs to identify code clones). We do not discuss these approaches in more detail as the general ideas were already presented and hybrid techniques only exploit them in combination.

Model Clone Detection Unlike source code, models are typically represented as visual elements (e.g., states and transitions) instead of a linear stream of statements in a textual form. Thus, the problem of identifying model clones (i.e., common parts) in compared models is an NP-complete

problem as it involves finding the largest common sub graph between the compared graphs [GJ79]. To cope with this complex problem, a number of algorithms exist that use different heuristics.

In general, model clone detection algorithms can be categorized into:

- *Graph-Based Approaches:* Using graph representations as an underlying structure to identify clones in the analyzed models.
- *Tree-Based Approaches:* Using tree representations of the analyzed models to identify clones.
- *Textual Approaches:* Using textual representations of the analyzed model to identify clones.
- *Statistical Approaches:* Using IR techniques and statistical clustering to identify clones in the analyzed models.

In the following paragraphs, we give an overview of corresponding algorithms.

Graph-Based Model Clone Detection Deißenböck et al. [DHJ+08] describe a graph-based approach for clone detection on flattened Matlab/Simulink models (i.e., all hierarchy levels introduced by subsystems are inlined). The approach uses normalized labels to abstract from irrelevant model information (e.g., the color or positions of model blocks) and to transform all blocks and edges to a graph structure. By creating all possible clone pairs and extending them with other clone pairs in the direct neighborhood, the algorithm tries to identify larger clones. Similar to our approach, the authors use a similarity metric to calculate structural similarity of blocks by not only analyzing the normalized graph labels but also the blocks' neighborhood. Overall, the authors are able to identify cloned model parts in Matlab/Simulink models and only present clones fulfilling certain criteria to the user (e.g., clones of a specific size) to not clutter the output. The algorithm was included in the Continuous Quality Assessment Toolkit (ConQAT) tool[10] and directly compared to the eScan algorithm by Pham et al. [PNN+09] in continued work on improving the relevance (e.g., adding improved filtering metrics), scalability (e.g., by removing clones found for a subsystem except for one prior to the clone detection) and tool support (e.g., with tooling to inspect clones) for the algorithm [DHJ+10]. Based on the labeled graphs generated by the infrastructure in ConQAT (i.e., the same graphs as in [DHJ+08]) Liang et al. [LCC14] propose their Optimized Path-Based Model Clone Detection (OPMCD) algorithm. Starting from graph nodes without incoming edges, the algorithm identifies paths longer than a specific threshold in the models and uses longest common subsequences between these paths to identify cloned parts.

Pham et al. [PNN+09] describe a similar graph-based clone detection approach included in their ModelCD tool that is also capable of identifying near-miss clones (i.e., subgraphs with small differences). Their eScan algorithm is focused on identifying exact clones and uses so-called canonical labeling functions assigning labels to sub graphs. Due to the characteristics of such labels (i.e., the same label is calculated for isomorphic subgraphs), it is possible to identify exact clones by checking for equivalent labels. In contrast, the authors' aScan algorithm uses a vector-based abstraction for compared model graphs called Exas [NNP+09]. By assigning a corresponding vector to each potentially cloned sub graph identified in the model graph and applying a distance measure, the algorithm is capable of identifying such near-miss clones.

[10] https://www.cqse.eu/en/products/conqat/overview/

Al-Batran et al. [ASH11] describe an approach to identify semantic clones in Matlab/Simulink models by applying semantic-preserving model transformation rules to achieve a normalized form of the compared models. These rules are based on mathematical (e.g., commutative operations), logical (e.g., De Morgan's Laws) and structural (e.g., replacing a multiplication with a constant by a corresponding *Gain Block*) semantics of Matlab/Simulink models to guarantee an unchanged behavior of the transformed models. Afterwards, the authors apply the algorithm from [DHJ+08] to the transformed models and, thus, detect semantic as well as syntactic clones. The authors claim that their approach is applicable to other dataflow formalisms and could be adapted to other languages (e.g., statecharts) applying similar ideas.

Störrle [Stö10, Stö13, Stö15] identifies during experiments that UML models represent loosely connected graphs with most of the models' information stored in their nodes. Based on this observation, the author develops an similarity-based comparison algorithm for nodes under their transitive closure. To reduce the overall complexity of the algorithm, filtering is applied to compare less graph nodes (e.g., by only comparing elements having the same meta class). The algorithm is implemented in the MQ_{lone} tool, which translates the *XML Metadata Interchange (XMI)* representation (i.e., a special *Extensible Markup Language (XML)* format used for exchange of models) of analyzed UML models to Prolog code for the executed clone detection.

Tree-Based Model Clone Detection Rattan et al. [RBS12] describe a generic approach to execute model clone detection for UML class diagrams. By applying standard XML parsing techniques, the authors create XML tree representations from the input models' textual XMI representation. By comparing sub trees of the imported models, the authors are able to identify clones based on found similar elements. As XMI is a common exchange format, the authors claim that their approach is applicable to other languages using the same format.

Liu et al. [LMZ+06] describe an approach to translate UML sequence diagrams to suffix trees and apply clone detection to this representation. By using an algorithm to detect the longest common prefix of suffixes in the created trees, the authors identify duplications in the compared sequence diagrams. Due to the design of the algorithm, the identified cloned parts are guaranteed to be extractable and, thus, can be used to refactor duplicated parts of UML sequence diagrams.

Textual Model Clone Detection In addition to the previous approaches, standard code clone detection can be applied to textual representation of the models allowing detection of cloned parts as well as near-miss clones. Alalfi et al. [ACD+12a, ACD+12b] describe an approach to execute text-based clone detection for Matlab/Simulink models by parsing the *mdl*-files[11] storing a textual representation of developed models. The authors use a reverse-engineered TXL grammar for the *mdl*-format to parse corresponding input files. However, in contrast to code artifacts, the order of model artifacts serialized in the *mdl*-files does not change the graphical semantic of the visual models. To account for these particularities, the SIMulink cloNE (SIMONE) detector (based on the NiCad near-miss code clone detector [RC08, CR11]) applies filtering and sorting methods on the textual model representation. Using this approach, the authors are able to identify model clones on different granularity levels (i.e., model, subsystem and block level) in Matlab/Simulink models. Based on the results, Cordy et al. [CDA14, CDA16] extend the approach for Matlab/Stateflow by extending the TXL grammar for the *mdl*-format and apply additional methods to cope with partic-

[11] Note, that the *mdl*-format was replaced by the XML-based *slx*-format in Matlab/Simulink 2012a [Pat14].

ularities of statecharts (e.g., splitting of transition labels into names, events, conditions and actions) and enable a meaningful comparison. These results allow the authors to execute clone detection of combined Matlab/Simulink and Matlab/Stateflow models (i.e., models involving blocks containing statecharts) to improve the overall accuracy of SIMONE [CDA16].

Similarly, Antony et al. [AAC13] describe how sequence diagrams can be analyzed based on their textual XMI representation, which is normally used to exchange models between different modeling tools. By applying a TXL grammar and specific source transformations the authors transform the input to normalized (i.e., identifiers are normalized) and contextualized self-contained (i.e., referenced elements scattered across the XMI file are inlined) model elements. Applying their near-miss code clone detector NiCad [RC08, CR11] the authors are able to identify not only exact clones, but also renamed and near-miss clones in sequence diagrams.

Statistical Model Clone Detection Babur [Bab18] describes an approach to identify (meta-)model clones (e.g., to assure the quality of DSLs). The approach is based on the author's Samos framework [Bab16, BCB16, BCV+16, BC17, BCB18] and extends its capabilities of applying statistical methods to (meta-)models (e.g., to identify clusters of related (meta-)models). By implementing additional facilities (e.g., to scope the identified clones to the level of meta-model classes or to extract the IR-features required by the clone detection), the author adds support for detection of clone clusters (i.e., containing more than one meta-model). These clones afterwards can be manually explored and analyzed to gain a deeper understanding of the clones.

Despite their capability of identifying reliable information on cloned parts for software artifacts, the discussed algorithms lack support for identifying the artifacts' differences. As a consequence, clone detection algorithms are not directly applicable for variability mining, because these differences are crucial for creating a 150% representation of compared artifacts and migrating to an SPL.

4.8.2. Differencing Techniques

Following the definition by Hunt et al. in their technical report on the program diff [HM76], differencing reports the differences between compared software artifacts in form of a minimal list of necessary changes to align the contents of the files. Differencing techniques rely on matching equivalent or near-miss parts prior to analyzing the differing parts of the analyzed software artifacts.

Source Code Differencing For our discussion of source code differencing techniques, we use the same categories as used by [RC07] in their survey on source code clone detection (cf., Section 4.8.1).

Text-Based Source Code Differencing Examples for *text-based source code differencing* are algorithms by Hunt et al. [HM76], Miller et al. [MM85] and Myers [Mye86]. For instance, Miller et al. [MM85, Mye86] use an algorithm to identify the shortest edit script between two text files. While the approach internally relies on a graph-based solution, we added it to this category as it uses textual comparisons of the files' lines and does not use PDGs, ASTs or tokenization. First the algorithm translates the compared files to a graph where the x-axis represents the lines of the first file and the y-axis represents the lines of the second file. The (x, y)-coordinates of the graph represent executed edit steps between the files. Horizontal lines represent removal of lines from the first file, vertical lines represent the addition of a line from the second file and diagonal lines represent parts where the lines are equal and no modification is needed. By searching the shortest path (i.e., the shortest edit script) through this graph, the algorithm identifies the differences between compared files.

Token-Based Source Code Differencing An example for *token-based source code differencing* is the algorithm by Lin et al. [LXX+14]. Lin et al. [LXX+14] describe an approach to identify differences between multiple reported clones (i.e., from different code artifacts) and, thus, to show developers refactoring potentials. By transforming the reported cloned code fragments from a clone class (i.e., a group of related clones) to tokens and applying longest common subsequence calculation to them, the authors first match all related tokens. The remaining tokens (i.e., the differences) are then compared to identify similar differences across the compared code artifacts. The algorithm is implemented in the tool Multi-Clone-Instance Differencing (MCIDiff).

Tree-Based Source Code Differencing Examples for *tree-based source code differencing* are the algorithms by Yang [Yan91], Chawathe et al. [CRG+96], Raghavan et al. [RRL+04], Falleri et al. [FMB+14] and Fluri et al. [FWP+07]. For instance, Falleri et al. [FMB+14] describe their GumTree algorithm to identify edit scripts (i.e., add, delete and update operations representing the differences) between the compared code artifacts. Their goal is to identify edit scripts that conform to actual modifications applied to the code rather than shortest edit scripts (i.e., the authors try to capture the developer's intent during modifications). By first applying a greedy top-down algorithm to ASTs of the compared code artifacts, the authors identify the largest unchanged code fragments as candidate mappings. Afterwards, the algorithm matches other cohesive pieces of code with a large number of common children based on these candidate mappings in a bottom-up manner. In case parts of this code are still unmatched, the authors apply an algorithm to identify the shortest edit scripts reflecting the differences between the compared code artifacts.

Program-Dependence-Graph-Based Source Code Differencing Examples for the *program-dependence-graph-based source code differencing* are algorithms by Horwitz [Hor90] and Binkley et al. [BCR+01]. For instance, Binkley et al. [BCR+01] describe an algorithm that uses slicing on PDGs (i.e., calculation of statements that affect a computation at a specified node in the PDG). These slices allow the authors to identify points in the compared programs that represent potential semantical differences. The extracted differences are used to identify relevant parts for regression testing and to elaborate on the potential effort reduction when concentrating on such differences during testing.

Model Differencing In addition, algorithms exist to identify the differences between modeling artifacts [SC12, SC13, KDP+09]. For our discussion, we use the same categories as for the model clone detection (cf. Section 4.8.1).

Graph-Based Model Differencing Lin et al. [LGJ07] describe a generic algorithm that identifies differences between domain specific models in their tool DSMDiff. Similar to our approach, the algorithm regards such models as graphs and traverses them in a DFS on a generic graph representation as nodes and edges. First, the algorithm tries to identify matches between the nodes by comparing their signatures (i.e., a string representation encoding their relevant information) as they are regarded most significant in models. Afterwards, the algorithm continues with matching the edges. For the mapped nodes and edges, the algorithm identifies differences based on their attributes. All unmatched elements are considered as differences (i.e., deletions and additions).

Tree-Based Model Differencing Kelter et al. [KWN05, KKP+12] describe their generic and adjustable SiDiff algorithm[12] to identify differences in models. Similar to our approach, the SiDiff algo-

[12] http://www.sidiff.org/

rithm relies on weighted similarity values calculated for compared elements and matches them only if their similarity is above a certain threshold. By first matching elements from the compared models that have the same hash value, the algorithm reduces the search space. Afterwards matching is executed in a bottom-up manner to match all elements from one model that have a unique match in the other model. The remaining model elements are matched iteratively in a top-down manner by propagating matches to all child elements of the previously matched elements. Using a special search tree data structure S^3V, the authors are able to provide an improved performance in calculating differences for large models [TBW+07].

Xing et al. [XS05] describe their UMLDiff algorithm, which is capable of calculating the differences for versions of UML class diagrams reverse-engineered from object-oriented classes. The algorithm traverses the models in a top-down manner and assumes that large parts of the compared models did not change. Thus, the unchanged model elements (e.g., classes) in each model serve as reference points during the comparisons and allow the authors to identify moved parts. Similar to our approach, the authors use similarity metrics to compare the names and structure of elements (e.g., the methods or constructors modifying a class field).

By EMF Compare[13], there is also support for model matching and differencing in the Eclipse platform [BP08]. The generic matching and differencing algorithms provided by EMF Compare can be exchanged with custom implementations of corresponding algorithms. The generic implementation is based on metrics that compare model elements and traverses the compared models synchronously as the authors assume that elements are seldom moved outside of their neighborhood (i.e., the model elements surrounding them). During the traversal, lists of matches are maintained and the algorithm is able to derive corresponding differences between the models.

Semantic Model Differencing Based on their SiDiff tool, Kehrer et al. [KKT11, KKO+12] describe their SiLift algorithm to semantically lift model differences to summarize low-level edits of models (e.g., add and remove operations during an attribute pull-up operation) in high-level changes better understandable by users (i.e., in a single change comprising the executed low-level changes). Furthermore, this semantic lifting is available for textual languages as the authors extend SiLift with support for textual models (e.g., DSLs that were developed with model-based techniques) [KPK+15].

Maoz et al. [MRR11a, MRR11b] describe with ADDiff and CDDiff two semantic differencing operators for UML activity diagrams [MRR11a] and UML class diagrams [MRR11b], respectively. By executing a search of semantic *difference witnesses* in activity diagrams, the authors are able to identify execution traces that are only contained in one of the compared activity diagrams. Furthermore, the authors transform compared class diagrams to Alloy[14] [Jac12], a textual modeling language that is based on relational first-order logic. By using the Alloy Analyzer [Jac12], the authors are able to identify semantic difference witnesses showing differences that are expressible with one of the compared class diagrams but not the other.

Statistical Model Differencing Babur [Bab16] describes an approach to identify (meta-)model clusters and outliers. The approach is based on the author's Samos framework [Bab16, BCB16, BCV+16], which provides statistical methods to analyze (meta-)models. Using different IR-techniques and different statistical methods, this framework allows clustering of (meta-)models showing the results

[13] https://www.eclipse.org/emf/compare/
[14] http://www.alloytools.org/

by means of dendrograms. By analyzing these dendrograms, it is possible to identify clear outliers (i.e., (meta-)models that are not related to the other (meta-)models). However, for an analysis of differences between (meta-)models, it is most interesting to explore and analyze the minor differences between related models as suggested by the dendrograms.

In addition to the discussed algorithms also commercial tools exist whose algorithmic details are not publicly available. For instance, DiffPlug[15] and SimDiff[16] are differencing algorithms targeting Matlab/Simulink models.

While the discussed differencing algorithms in general identify the commonalities (i.e., matched parts) and differences between compared models, they do not aggregate them in a unified representation and, most importantly, do not classify the variability in mandatory, alternative and optional parts. As a result, these approaches are not applicable for fine-grained variability mining of related variants, or at least not without additional effort.

4.8.3. Hybrid Techniques

Merging related software artifacts has been a research topic for quite some time and a number of approaches exist for source code [ALL+17b, Buf95, CW98, ELH+05, GM13, Men02] and models [ALL+17b, ASW09, LAD+17, LC13, SC12, SC13, SW17b]. Clearly, hybrid techniques, such as merging in the context of *Version Control Systems (VCSs)* or reverse-engineering variability, exploit both dimensions of variability (i.e., the common and differing parts). However, depending on their intention, not all existing approaches are designed to explicitly identify and annotate the identified *variability* in the merged artifacts. For example, algorithms for merging artifacts in classic VCSs are designed to align and merge changed parts of the compared artifacts, but do not explicitly categorize and annotate the identified variability. Thus, we distinguish between approaches that execute merging in a *variability-agnostic* manner and others that are *variability-aware* and add corresponding annotations to the merged artifacts.

Variability-Agnostic Hybrid Techniques for Source Code Variability-agnostic hybrid techniques for source code can mainly be found in the context of VCSs to allow distributed development of functionality and merge results into a single repository [Buf95, CW98, ELH+05, GM13, Men02]. Examples are approaches by Tichy [Tic82, Tic85], Westfechtel [Wes91] and Rochkind [Roc75], but also algorithms integrated in commonly used VCS tools, such as Apache Subversion (SVN)[17] or git[18]. For instance, Tichy describes for the GNU Revision Control System (RCS)[19] that the tool diff can be used to calculate the differences between revisions of files in form of edit scripts. By always storing the newest revision of the edited file together with backward edit scripts (i.e., differences allowing to restore previous versions when applied to the newest version), the author is able to restore previous versions of a file. In addition to these algorithms for VCSs, other approaches exist that focus on merging source code. For example, Higo et al. [HKI08] and Jarzabek et al. [JL03, JL06] focus on reducing potential negative effects of code clones by identifying opportunities to merge them or applying meta-programming techniques, respectively. Higo et al. [HKI08] identify code clones using standard clone detection algorithms and, afterwards, calculate metrics for clones in

[15] https://www.diffplug.com/features/simulink
[16] http://www.ensoftcorp.com/simdiff/
[17] https://subversion.apache.org/
[18] https://git-scm.com/
[19] https://www.gnu.org/software/rcs/rcs.html

the same clone classes (e.g., whether they are contained in the same class and how much dependencies exist to parts outside of the cloned code). Based on the calculated values, the authors indicate possibilities to remove the clones by merging them (e.g., by using the pull up method strategy to move common code to a base class). Jarzabek et al. [JL03, JL06] apply meta-programming to develop classes with common code that has slight modifications. Using the *XML-Based Variant Configuration Language (XVCL)*, the authors are able to express variability in classes by means of meta-constructs (e.g., meta-variables allowing to be replaced by a type that is processed by a method) and to generate different implementations from such descriptions.

Variability-Aware Hybrid Techniques for Source Code In addition to variability-agnostic techniques, a variety of approaches exists to reverse-engineer variability relations for source code [ALL+17b, LC13].

Alves et al. [AMC+05, AGM+06, AMC+07, ACN+08] describe an approach to extract variability from existing variants and transfer it to an AOP SPL with additional refactoring support to evolve the created SPL (e.g., to accommodate new variants). By providing support in their tool suite FLiP, the authors enable developers to manually select variability that should be extracted into AOP aspects according to refactoring rules provided by FLiP. Furthermore, the authors provide means to react to new requirements by incorporating new variants and evolving the created SPL.

While the authors identify fine-grained variability, their approach is limited to languages with AOP implementations without providing tooling for a corresponding adaptation. In addition, the approach only allows a tool-supported manual transition to an SPL. In contrast, we provide an automated approach, that only requires manual interaction in case of conflicting matches without selected automated resolution strategies. Furthermore, we provide tooling to support developers during adaptation of our approach for new languages.

Klatt et al. [KK12, KK13, KKK13, KKS14, KKW14, Kla14] describe the tool-supported SPLevo approach to semi-automatically migrate related product variants to an SPL. By first automatically calculating matching and differencing artifacts, the authors identify the commonalities and differences between the compared source code variants [KKK13, KKS14, Kla14]. Based on these results, the authors initialize a variation point model that can be used by the executing developers to iteratively group and merge variation points using automatically derived recommendations [KKK13, KKS14, Kla14]. These recommendations are calculated using a dependency analysis between the identified variation points (e.g., considering PDGs, cloned changes or program execution traces). Using the identified variation point model and SPL profiles (i.e., settings of the created SPL) defined prior to the migration, developers can manually implement the SPL for the analyzed variants.

The approach by Klatt et al. [KK12, KK13, KKK13, KKS14, KKW14, Kla14] is similar to ours as it uses an intermediate format for the representation of identified variability relations (i.e., variation point models vs. 150% models). These models allow detailed analysis of the identified variability and manual adjustment if desired. In addition, both approaches strive for adaptability by using a generic structure (e.g., PDGs for Klatt et al. and the base meta-model for our approach) but allowing language-specific analysis steps (e.g., the dependency analysis with specific rules for Klatt et al. and language-specific metrics for our approach).

Linsbauer et al. [FLL+14, FLL+15, LFL+15, LLE17] describe algorithms to extract the variability of product variants by means of feature traces from artifact trees (e.g., ASTs). By comparing these artifact trees, the authors identify implementation artifacts realizing the upfront known features of products and corresponding dependencies between them.

The authors are capable of identifying variability between variants in form of complete features. However, they do not provide fine-grained variability information to analyze explicit variability relations between variants (i.e., whether elements are mandatory, alternative or optional). In contrast, we allow analysis of the fine-grained variability relations by means of 150% models and transfer this information to coarse-grained artifacts in subsequent steps.

Fenske et al. [FMS+17] use incremental variant-preserving refactorings to transition a set of cloned variants step-by-step into an SPL. The authors identify the common parts of variants using a clone detection tool and try to move them to the common core by using the pull up method strategy. For detected *Type 2 Clones* (e.g., clones with renamed variables), the authors apply refactorings to align the used names and, thus, try to eventually move them also to the common core. Overall, the authors enable developers to create a common core by iteratively applying the refactorings and to extract the variability in form of FOP features.

The approach by Fenske et al. [FMS+17] is capable of identifying and aligning variability between variants, which can be adapted for other languages (e.g., also for model-based languages). However, it only provides tool-supported manual techniques to migrate the variability to an SPL and requires the engineers to select parts that should be extracted. In contrast, we automatically migrate analyzed variants to an SPL (cf. Chapter 5).

Ziadi et al. [ZFS+12] use the notion of *interdependent atomic model elements* to identify cohesive functionality. By transforming object-oriented source code to a UML class diagram and translating this representation to atomic model elements (e.g., classes, fields and methods), the authors abstract from the actual source code. Afterwards, the approach uses an interdependence relation over the identified set of atomic model elements to group them into features (i.e., atomic model elements forming equivalence classes across the atomic model elements of all compared models). In continued work, Ziadi et al. [ZHP+14] extend this approach and apply the identification of interdependent model elements to ASTs of the complete product variants. Based on this code representation, the authors' ExtractorPL tool is capable of deriving *Feature Structure Trees (FSTs)* – i.e., ASTs for features – for the identified features and generate different product variants based on these artifacts.

The approach by Ziadi et al. [ZFS+12] uses generic and language-independent techniques to identify features between compared variants (i.e., the interdependent model elements). However, executing the analysis on such a generic level only allows to identify coarse-grained variability relations. In contrast, our approach uses a generic comparison level and hands detailed analysis over to user-adjustable metrics allowing to identify fine-grained explicit variability relations between variants (i.e., whether elements are mandatory, alternative or optional).

Nöbauer et al. [NSG14] describe an approach to provide variability mechanisms to companies that implement extensions of standard software for their customers (i.e., software developed by external companies). The authors assume that traceability between requirements (stored in a database) and the implementation artifacts exists. This enables their approach to identify different implementations of specific requirements and to derive a corresponding feature model with requires and parent-child relations. Upon detecting changes in the requirements database, the feature model is updated. All identified information can be refined by developers and, for example, a meaningful variability has to be assigned to features as it is not identified (by default all features are optional).

The approach by Nöbauer et al. [NSG14] relies on traceability of requirements to corresponding implementation artifacts for feature identification. In contrast, our approach is capable of iden-

tifying variability between product variants without explicit knowledge of requirements for their implementation. However, if available, such information could easily be integrated in the metrics used for concrete comparisons between variants.

Reinhartz-Berger et al. [RZW16] describe an approach to identify variants and suggest variability realization mechanisms based on ontological relations between SPLs and software variants. The approach is based on previous work by the same authors, which introduced a list of ontological categorizations for variability mechanisms (i.e., how to model variability) of software variants [RZW15]. By extracting behavioral descriptions from PDGs and structural information from class diagrams, the authors identify and cluster similar parts of the compared variants and suggest to apply variability realization mechanisms based on their ontology.

While the authors are only able to suggest possible variability realization mechanisms for the identified variation points by means of their ontology and do not automatically merge the variability, we also allow for an actual migration to an SPL (cf. Chapter 5).

Méndez-Acuña et al. [MGC+16a, MGC+16b] describe an approach to identify reusable language modules across different DSLs relying on meta-modeling techniques. By comparing a set of DSLs based on their meta-models and identifying overlapping parts, the authors identify reusable language parts and extract them based on the graph representation of the compared meta-models. The resulting parts are encapsulated in reusable modules and can be shared across the analyzed languages. The authors implemented their approach in the Puzzle tool [MGC+16a].

While this approach is limited to enabling a module-based reuse of language capabilities across different DSLs, our approach aims at actually migrating related variants to an SPL.

Variability-Agnostic Hybrid Techniques for Models Similar to source code, the larger part of existing variability-agnostic hybrid techniques for models can be found in the context of VCSs [ASW09, SC12, SC13, SW17b]. Examples are approaches by Mehra et al. [MGH05], Oda et al. [OS06], Alanen et al. [AP03], Murta et al. [OMW05, MCP+08, MOD+07] and Barone et al. [BDF+08, DFS+09]. For instance, Barone et al. [BDF+08, DFS+09] describe an approach that considers each model entity separately for versioning and computes their differences using the UUID of the entities. The identified differences are sent to the VCS server and the entities can be added (i.e., no matching element previously existed), removed (i.e., the entity with an existing UUID was deleted) or merged (i.e., changes to model entity attributes were modified). In addition to these approaches, other algorithms exist to merge descriptions of behavioral models that were created by experts with different (partial) views on the modeled system (e.g., Uchitel et al. [UC04]), to merge behavior expressed by multiple statecharts into a single statechart representation (e.g., Frank et al. [FE99]) or to generate statechart implementations by merging sets of sequence charts that were semantically enriched to detect and resolve potentially conflicting descriptions (e.g., Whittle et al. [WS00]). Furthermore, Kolovos et al. [KPP06a] propose the *Epsilon Merging Language (EML)* which is based on the authors' *Epsilon Object Language (EOL)* [KPP06b] and allows rule-based implementation of model merging.

Variability-Aware Hybrid Techniques for Models In addition to variability-agnostic approaches, also techniques exist to reverse engineer variability for models [ALL+17b, DP09, LC13, SC12, SC13, SW17b].

Alalfi et al. [ACD14, Cor13] cluster identified subsystem clones based on the results in [ACD+12a, ACD+12b] (cf. Section 4.8.1 – *Textual Model Clone Detection*) to identify clone patterns. Based on subsystem clone clusters, the authors apply a simple graph matching strategy to identify common

blocks between the clones and infer that non-cloned subsystem parts represent variability [ARS+14]. The authors categorize the identified variability by five Matlab/Simulink-specific categories, annotate them to the clones and merge these variation points directly in a single Matlab/Simulink model by means of *Variant Subsystem Blocks* (i.e., *Subsystem Blocks* allowing selection of the executed behavior). These results are visualized in their SimNav [RSA+15] extension for SIMONE.

While the approach by Alalfi et al. [ARS+14] is capable of identifying variability in Matlab/Simulink models, it relies on the SIMONE clone detection approach for Matlab/Simulink models and, thus, is not directly applicable to other languages. Furthermore, SIMONE provides only support to adjust settings concerning the clone detection (e.g., thresholds for near-miss clone detection). As SIMONE uses NiCad as an underlying clone detection technique, the applied ideas, in theory, could be realized for other languages by integrating corresponding TXL grammars. However, no tooling exists to support developers during such an adaptation. Furthermore, the used variability categories are Matlab/Simulink-specific and, thus, are not generic enough to be applicable for other languages.

Zhang et al. [Zha10, ZHM11, ZHM12, Zha14] introduce the CVL Compare approach for MOF-based models, which abstracts from low-level variability identification (e.g., on a state level) by encapsulating the low-level variability in larger variation points. The authors use facilities provided by the Eclipse-based EMF Compare tool (cf. Section 4.8.2) to identify the common and varying parts of the compared models. Starting from a user-selected base model, the authors identify for each remaining model the common and varying parts with regard to the base model. Based on these results, the authors create a preliminary CVL feature model by executing higher-order comparisons on the identified low-level differences to only contain elements that were not previously considered in the feature model. The created features refer to variation points (i.e., placement fragments) in the base model, which can be replaced with different model elements (i.e., replacement fragments). The resulting feature model can be manually refactored by applying domain knowledge to create a final product line meeting the domain experts' expectations. In continued work, the authors describe how to incorporate additional variants into the created product line [ZHM12].

The approach by Zhang et al. [ZHM11] is similar to ours as the authors (semi-)automatically extract variability from a set of existing product variants. Zhang et al. [ZHM11] use the facilities provided by EMF Compare and, thus, allow generic variability mining. However, they completely rely on these algorithms with only limited possibility to influence how the models are traversed and elements are compared (cf. Section 4.8.2). In contrast, our algorithms exploit information about the execution-flow in the models to reduce the overall number of executed comparisons. Compared to Zhang et al. [ZHM11], our approach additionally allows domain experts to involve their knowledge about the compared models directly in the comparisons (i.e., by creating a custom-tailored metric) and not only at the end of the process when refactoring the final SPL. In addition, as we generate a 150% model of all variability as an intermediate result, our approach is not limited to creating CVL SPLs, but allows translating the results to different concrete SPL realizations (e.g., CVL or DOP).

Font et al. [FBH+15] introduce their Model Family to SPL approach to transition a number of related model variants to a CVL-based SPL. First, the authors calculate all differences for all pairwise combinations of the considered model variants using EMF Compare. Based on the corresponding results, the authors select a base model that has the lowest number of differences to be addressed during transformations to the other models. Using the previously identified differences, the au-

thors define variation points over placement fragments in the base model and select corresponding possible replacement fragments from the other models to create a model library of fragment substitutions. In addition, the approach creates a resolution model representing the necessary configurations of all variation points to derive each model variant involved in the comparison. Based on the tool support for their approach, the authors allow developers to create new variants and incorporate them by automatically executing their approach to identify new parts missing in the existing SPL. During their evaluation, the authors identify five scenarios of modifications to the SPL (e.g., addition of new placements or replacements).

The approach [FBH+15] is highly similar to the one by Zhang et al. [ZHM11] as it relies on the same technologies and is only extended with an automatic base model selection and a classification of five SPL evolution scenarios. Thus, the same differences to our Family Mining apply.

Martínez et al. [Mar16, MZK+14] introduce an approach called Model Variants Comparison (MoVaC) to identify and visualize variability in sets of MOF-based models. The approach relies on the notion of *interdependent atomic model elements* introduced by Ziadi et al. [ZFS+12] to identify cohesive functionality. For the approach by Martínez et al., these atomic elements are generated by traversing the given models and considering the language-independent class, attribute, and reference information provided by the Ecore meta-model. Afterwards, similar to Ziadi et al. [ZFS+12], Martínez et al. [MZK+14] use an interdependence relation over the identified atomic model elements to group them into features. To this end, the authors use a similarity metric to identify equivalent atomic elements. Martínez et al. [MZB+15a] extend their work from [MZK+14] and describe how the identified features can be used to transfer the variability of the compared product family to a CVL-based SPL. By analyzing the dependencies ot the atomic elements used to identify the features in the set of compared products, the authors identify *requires* and *excludes* dependencies. The CVL-based SPL is realized by creating a 150% model containing all identified features and together with negated versions of the same features (i.e., descriptions how to remove them). Thus, this maximum 150% model contains all possible features and represents a subtractive SPL.

The described algorithms from [MZK+14, MZB+15a] are implemented in the extensible and generic framework Bottom-Up Technologies for Reuse (BUT4Reuse) [MZB+15b, MZB+17]. It provides different adapters to allow extension of the framework (not only for modeling languages, but also textual languages or even Eclipse plug-in structures) with algorithms to handle different file types, to introduce comparison algorithms, to identify and locate features as well as constraints between them and to export reusable artifacts.

While the approach by Martínez et al. [MZK+14, MZB+15a] is meta-model independent as it makes use of the structures provided by the MOF, it only identifies high-level variability relations (i.e., features) between the compared variants. Thus, fine-grained analysis of the variability relations between the product variants on a low level is not possible as they are abstracted by the created features. In contrast, our approach allows experts to identify explicit low-level variability relations (i.e., whether elements are mandatory, alternative or optional) between product variants, analyze them and transfer this information to an SPL (cf. Chapter 5). Similar to our approach, the authors' framework provides adapters to allow adaptation of the provided algorithms and to include custom algorithms. However, compared to our approach, the authors do not provide additional tooling to support developers in adapting their approach for new languages (e.g., to create similarity metrics for new adapters).

Rubin et al. [Rub14, RC12, RC13d] formally specify the *merge-in* operator that compares, matches and merges related software variants. The operator uses a similarity-based approach to compare all model elements with each other and identifies matches based on these similarities. Based on these results, all varying parts are merged into a single model with annotations showing which parts belong to the different model variants. This approach is executed in an iterative pairwise manner to merge all compared models into a single model. Applicability of this formal operator is shown by an implementation for UML class diagrams and statecharts. In continued work, the authors demonstrate that early merge decisions in pairwise approaches can result in inadequate merges in later iterations (e.g., a better match could have been created by another relation, but is prevented by the earlier merge) and, thus, unexpected variability relations for the domain experts [RC13c]. To solve these problems, the authors propose the *n-way model merging* that creates and chains tuples of model elements from all compared models. Using a custom heuristic algorithm for the NP-hard weighted set packing problem, the authors identify optimal solutions to maximize the overall similarity for merged models. Applicability of the approach is demonstrated for UML class diagrams.

In additional work, Strüber et al. apply the general ideas of merging variability from [RC12, RC13d] to model transformation rules [SRA+16]. The approach uses clone detection techniques to identify common parts across the rules and clustering to find rules that yield potential to be combined into variable transformation rules. By first matching the common parts of variable rules and afterwards evaluating their differing parts, the overall rule matching effort can be reduced and better performance can be achieved compared to single transformation rules [SRC+15, SRA+16].

While the approach of Rubin et al. in [RC12] is highly similar to our Family Mining (i.e., the similarity based comparison and steps executed during variability mining), it executes comparisons between *all* model elements of a specific type (e.g., UML classes). In contrast, our approach aims at reducing these comparisons by exploiting the execution-flow of analyzed models. This approach can reduce the execution time for models dramatically as less relations have to be analyzed. Furthermore, the approach by Rubin et al. is only demonstrated for UML class diagrams [RC12, RC13c] and statecharts [RC12], while we support adaptation of our approach for new languages (e.g., by using the Vampire DSL and a generic base meta-model). Also, the similarity metrics used by Rubin et al. in [RC13c] are less flexible than ours as they only allow for binary similarities of the compared UML classes. For fine-grained comparison of other languages (e.g., Matlab/Simulink and statecharts), such binary relations are not sufficient as relative similarities might be necessary to consider the complete information (e.g., only one of two transition events might match). However, the chaining algorithm relies on this property of the metric and additional steps would be necessary to use it with relative similarities. In addition, no explicit variability relations (i.e., whether elements are mandatory, alternative or optional) are identified and variability is only modeled by all elements concerning a specific variant (i.e., a single feature represents each variant).

Boubakir et al. [BC16] merge a set of UML models in a pairwise manner. The authors compare and calculate a similarity value between all model elements from all compared models and use algorithms to optimize the solution to identify the best match. Based on these results, the authors iteratively merge compared models with the highest similarity to find an optimal solution.

The authors' approach is highly similar to techniques presented by Rubin et al. in [RC12] and [RC13c]. It mostly differs in the additional calculation of overall similarities between merged models to optimize the overall solution. Similar to [RC13c], the authors use a metric to calculate binary

similarities (i.e., complete matches) between model element properties. Thus, it is less flexible than our approach and might not identify near-miss relations, such as renamed model elements. Furthermore, the algorithm compares each model element from one model with all model elements from all other models, which leads to combinatorial explosion of analyzed relations. While this approach might scale for sets of small models (i.e., the authors demonstrate applicability to rather small UML class diagrams), it will most certainly fail for sets of industrial-scale models (e.g., industrial Matlab/Simulink models with multiple thousand blocks per model). In contrast, our approach exploits the execution-flow in models to reduce the number of compared model elements.

Ryssel et al. [RPK10, RPK12a, Rys14] identify subsystem variability in Matlab/Simulink model variants and transfer them to a FOP SPL. The authors use a detailed similarity metric for the comparison of Matlab/Simulink models, which is used to create a similarity matrix of all model subsystems. Prior to executing the detailed variability analysis between compared models, the authors use the calculated similarity matrix to execute a subsystem clustering that is similar to our model clustering (cf. Section 3.4). The resulting subsystem clusters are used to create variation points of similar subsystems by building cliques of highly related model blocks and in turn so-called *model templates* (similar to our 150% models). In a subsequent step, the authors identify feature models for low-level variation points on the level of (grouped) blocks and connectors. These are in turn used to create a *Multi Software Product Line (MSPL)* – i.e., an SPL combining the functionality of multiple other SPLs – for compared models by merging the different subsystem feature models.

The similarity calculation executed by Ryssel et al. is similar to our approach. However, the authors execute the comparisons only on subsystem level to identify corresponding variation points and matching feature models. The authors rational behind this approach is based on the relevance of subsystems in Matlab/Simulink models as containers of coherent functionality. While the authors only demonstrate applicability of their approach for Matlab/Simulink models, the same ideas might be applicable for other languages using container entities similar to subsystems (e.g., regions in statecharts). However, no support for an adaptation exists. In contrast, our algorithm uses a more generic approach by comparing complete model variants and, thus, is able to support a wider range of different languages independent of such constructs. Furthermore, we provide means to adapt our algorithms for new languages (e.g., by using the Vampire DSL).

Sabetzadeh et al. [Sab08, SE03, SE06, SNE+07] merge models that might contain inconsistencies or are incomplete due to different conflicting views of stakeholders (e.g., whether certain parts should be present in the merged models). Thus, in contrast to other related approaches (e.g., by Pottinger et al. [PB03] or Buneman et al. [BDK92]), the proposed solution is capable of merging views that were not a priori aligned (e.g., by using the same data types or resolving naming conflicts). The authors use a category theoretic approach to define a three-way merge of such views by using interconnection diagrams. These diagrams allow to express relations between the merged models and to annotate the certainty of stakeholders involved. Based on this solution, it is possible to merge models and show annotations for parts where stakeholders disagreed on a solution.

While the approach by Sabetzadeh et al.+[SE03, SE06, SNE+07] is capable of merging models with commonalities and differences, it requires a manual definition of relations between the models. Furthermore, the merged models do not provide explicit variability relations (i.e., whether elements are mandatory, alternative or optional). Thus, the approach is not applicable for automated variability mining as the involved manual effort would be too high for large sets of models.

Nejati et al. [Nej08, NSC+07, NSC+12] match and merge related statecharts in scenarios where different viewpoints on the same feature are implemented (e.g., by different stakeholders). The authors calculate matchings similarities between states based on their static properties (e.g., their names) and behavioral properties (e.g., their incoming and outgoing transitions). By using `WordNet::Similarity` [PPM04] (i.e., to identify the semantic relatedness of terms), the authors account for potential differing terms that are used by developers for the same concepts (e.g., *shut* vs. *closed* for the status of a door). Afterwards, the resulting relations are merged into a single statechart representation with presence conditions added to elements only present in one of the variants.

While the approach by Nejati et al. allows merging of statechart variants, it is specifically tailored for statecharts with limited adjustability to different settings (e.g., adding support for comparisons of state actions as supported by ETAS ASCET). In contrast, our approach provides similar algorithms for a wide range of modeling languages based on a generic base meta-model and comparisons executed by user-adjustable metrics. Furthermore, our merged 150% models contain explicit variability information for relations (i.e., whether elements are mandatory, alternative or optional).

Pietsch et al. [PKK+15] describe their S1PL approach, which is based on the authors SiLift framework [KKT11, KKO+12, KKT13] to identify semantic model differences (cf. Section 4.8.2). The corresponding information is encoded in edit scripts that are directly considered as delta modules and have to be manually enriched with corresponding application conditions. Thus, the authors directly encode the identified variability in delta modules, while our approach first merges a 150% model making fine-grained variability explicit (i.e., mandatory, alternative or optional parts).

Variability-Aware VCSs In addition to these approaches, a number of solutions exist to support development of variability across product variants by means of variability-aware VCSs [LBG17].

For instance, **Schwägerl et al.** [SW16, SW17a] propose a VCS-based approach for evolving model-based SPLs. The approach allows to check-out revisions of SPL variants configured through the feature model provided by the VCS. After editing the selected variant, the authors assign the applied changes to the edited feature and make it available to future revisions upon commit.

In general, the approach is adaptable for other languages (through the generic VCS techniques) and capable of merging model artifacts considering the existing variability of an SPL. However, the approach allows to develop an SPL in a proactive or reactive manner (i.e., starting from scratch and reacting to new customer requirements), while our approach targets creation of an SPL in an extractive manner (i.e., mining variability from existing artifacts).

Semantic Variability Analysis based on Test Cases Another related approach is proposed by Richenhagen et al. [RRS+16] to support developers during SPL extraction. The authors translate existing test cases to automatons and compare them to calculate semantic similarities between the analyzed products. While these similarities allow guidance of developers during the migration to an SPL, the approach does not support developers with automatic migration steps (e.g., the extraction of differing functionality).

Variability Mining in Business Process Management Furthermore, also research on reverse engineering variability in the area of *business process management* exists [LAD+17]. Business process management is concerned with identifying optimization potential in business processes of companies [Aal16]. Examples for such processes are the manufacturing of products (e.g., assembling the engine of a car prior to installing it) or providing support to customers (e.g., the process triggered

when a customer files a complaint). To identify such processes, *process mining* can be applied to identify processes executed in a company by, for example, analyzing process trace data (e.g., event logs from executions) [Aal16]. Examples for variability mining in business processes are the approaches by La Rosa et al. [LDU+13], Assy et al. [AGD14] and Li et al. [LRW11]. For instance, Li et al. [LRW11] analyze variants of a business process model to identify an improved process that fits better to their common original reference model. By transferring the analyzed processes models to a matrix representation, the authors are able to discover an initial reference model or to identify an improved reference model learning from the original inputs.

While these approaches identify fine-grained variability between the processes and are configurable to adjust the results to some degree towards the users' expectations, they are specifically designed for business process mining. Thus, they are not applicable for other modeling notations.

Migrating Single Products to SPLs The described approaches concentrate on identify variability across multiple related variants. However, also approaches exist to refactor single program implementations into SPLs. Examples are the approaches by Liu et al. [LBL06] and Kästner et al. [KDO14].

For instance, **Kästner et al.** [KDO14, KDO11] describe a semi-automatic approach, where a human-in-the-loop expert provides seed fragments for features to a system giving recommendations for further associated code. Using a graph structure based on ASTs, the authors' Location, Expansion, and Documentation Tool (LEADT) removes code from the analyzed implementation that is associated with the feature seed fragment. During this process, the authors utilize a type system for the corresponding programming language to infer which parts of the code are related to the feature (e.g., by following references or usage relations) and to check if the code is compilable without the removed code. At this point, the tool derives recommendations to include the removed code in the feature and reports them back. Based on these results, the user can iteratively refine the seeds by excluding or including additional code.

While these approaches are capable of extracting features from single products and creating corresponding SPLs, they do not provide means to transition multiple related variants to an SPL. In contrast, our approach is able to analyze multiple variants and migrate them to an SPL (cf. Chapter 5).

Supporting Developers during Clone-and-Own In addition to approaches allowing migration to an SPL, also support for improved clone-and-own strategies is needed as companies tend to not migrate all variants at once or do not take this step at all (e.g., due to fear of the linked risks) [DRB+13, RC13a]. Different authors propose approaches to support development of software using clone-and-own strategies. Examples are the approaches by Rubin et al. [Rub14, RC13a, RCC13, RCC15], Pfofe et al. [PTS+16], Lapeña et al. [LBC16] and Fischer et al. [FLL+14, FLL+15].

For instance, Rubin et al. [Rub14, RC13a, RCC13, RCC15] provide support to maintain variants that were not yet integrated in a common SPL and to later migrate them to an SPL. To this end, the authors propose a framework consisting of conceptual operators that can be implemented by companies to support maintenance of cloned variants and their transition to managed reuse in an SPL over time. The operators allow, for example, to apply feature location techniques to identify source code for specific features or to show dependencies between features. Furthermore, Fischer et al. [FLL+14, FLL+15, LFL+15, LLE17] describe their Extraction and Composition for Clone-and-Own (ECCO) tool that allows to use quasi SPL techniques to generate new variants from existing clone-and-own artifacts. Based on their algorithms to extract feature traces and feature de-

pendencies from product variants, the authors build a database allowing to generate new variants by combining the extracted features into new variants. The approach also provides means to guide developers during implementation of additional glue code that might be needed (e.g., for feature combinations that previously did not occur) or new features. In addition, Lapeña et al. [LBC16] present their Computer Assisted Clone-and-Own (CACAO) approach, that applies NLP techniques and clustering on requirements to suggest potential relevant methods for developers during development of new products using clone-and-own.

These approaches provide support in clone-and-own scenarios where no transition towards an SPL or only a gradual one is possible. Thus, these approaches can be mainly seen as orthogonal to our Family Mining and can help developers to ease problems related during clone-and-own. However, despite targeting the transition to an SPL, our Family Mining approach also supports management of clone-and-own variants as it allows to identify their low-level variability. Based on the created 150% models, developers can easily analyze the variants' common and varying parts.

Recovering Variability Information from Software Architectures In addition to these approaches, algorithms exist to recover software architectures from developed software systems [DP09]. While the greater part of these algorithms are variability agnostic (e.g., [DK16, KC99]), there also exist approaches to reverse-engineer the software architecture of product families [DP09]. Examples are approaches by Pinzger et al. [PGG+03], Stoermer et al. [SO01], Koschke et al. [FKB+07, KFB+09], Acher et al. [ACC+11] and Shatnawi et al. [SSS15]. For instance, Acher et al. [ACC+11, ACC+14] reverse-engineer variability models of software architectures in form of feature models. By extracting a feature model for 150% models of the family's components and mapping this information to a feature model of the used plug-ins, the authors are capable to aggregate this information in architecture feature model expressing dependencies between the components (by means of constraints) and their possible configurations (by means of the feature variabilities).

While these approaches are able to identify high-level architectural variability from source code artifacts or system documentations, they do not provide fine-grained variability relations. As a result, these approaches are not applicable for variability mining of block-based models.

Summary All discussed approaches related to our Family Mining approach are summarized in Table 4.8. This table comprises details on the identified variability information (i.e., cloned and differing parts) and the merging capabilities. Furthermore, we show if the approach is able to identify coarse-grained and fine-grained variability for the analyzed variants and whether it is adaptable for new language and adjustable to user-defined settings. As we can see, a large variety of different approaches exist to identify variability information for source code and model variants. While *Clone Detection* and *Differencing* algorithms in general are capable of identifying relevant information, they lack support for merging of variability in a single representation. In addition, the existing *Variability-Agnostic Hybrid Algorithms* and *Software Architecture Variability Mining* approaches are capable of merging the variability information, but they are not able to identify fine-grained variability information. In contrast, *Business Process Variability Mining* is able to identify such information, but is focused on business processes (e.g., event logs) and not adaptable for other languages. The remaining approaches, which are directly related to our approach mostly lack support for all of our criteria and often only comprise a subset of them. Most notably, all of these approaches are not able to identify coarse-grained variability relations between the analyzed artifacts or use corresponding

		Cloned Parts	Differing Parts	Merge Artifacts	Coarse-Grained Variability	Fine-Grained Variability	Adaptability	Adjustability
Family Mining	M	+	+	+	+	+	+	+
Clone Detection Algorithms	D	+	–	–	o [Bab18]	–	o	o
Differencing Algorithms	D	+	+	–	o [BCB16]	–	o	o
Variability-Agnostic Hybrid Algorithms	D	+	+	+	–	–	o	o
Software Architecture Variability Mining	D	+	+	+	–	–	–	o
Business Process Variability Mining	M	+	+	+	–	+	–	o
Support for Clone-and-Own	D	+	+	+	o [LBC16]	–	o	o
Alalfi et al. [ARS+14]	M	+	+	+	–	+	–	–
Alves et al. [AMC+05]	SC	+	+	+	–	+	o no tooling	+
Boubakir et al. [BC16]	M	+	+	+	–	o not explicit	o no tooling	–
Fenske et al. [FMS+17]	SC	+	+	+	–	–	+	+
Font et al. [FBH+15]	M	+	+	+	–	+	+	o EMFCompare
Kästner et al. [KDO14]	SC	+	+	+	–	–	+	+
Klatt et al. [Kla14]	SC	+	+	+	–	+	+	+
Linsbauer et al. [LLE17]	B	+	+	+	–	–	+	–
Liu et al. [LBL06]	SC	+	+	+	–	–	+	+
Martínez et al. [Mar16]	B	+	+	+	–	–	o no tooling	+
Méndez-Acuña et al. [MGC+16b]	SC	+	+	+	–	o only modules	–	–
Nejati et al. [Nej08]	M	+	+	+	–	o not explicit	–	+
Nöbauer et al. [NSG14]	SC	+	+	+	–	–	+	–
Pietsch et al. [PPK+15]	M	+	+	+	–	o not explicit	+	–
Reinhartz-Berg et al. [RZW16]	SC	+	+	+	o variation points	o only partial	o no tooling	–
Rubin et al. [Rub14]	M	+	+	+	–	o not explicit	o no tooling	o only [RC12]
Ryssel et al. [Rys14]	M	+	+	+	o for subsystems	+	–	+
Sabetzadeh et al. [Sab08]	M	+	+	+	–	o not explicit	o no tooling	–
Schwägerl et al. [SW16]	M	+	+	+	–	–	+	–
Zhang et al. [Zha14]	M	+	+	+	–	+	+	o EMFCompare
Ziadi et al. [ZHP+14]	SC	+	+	+	–	–	+	–

Supported artifacts: *M* = models *SC* = source code *B* = both, models *and* source code *D* = depends

Table 4.8.: Comparison of our Family Mining approach in Chapter 4 with related approaches.

capabilities only internally to set subsystems or variation points into relation (i.e., [RZW16, Rys14]). Thus, all approaches implicitly seem to assume that such details are available in all companies. Especially in clone-and-own scenarios, we believe that this assumption is simply wrong, because clone-and-own is not executed in a structured manner and in most case is not documented at all [DRB+13]. As a result, these high-level relations are not available and have to be identified in tedious manual work. In contrast, our Family Mining approach in combination with our Corevid approach (cf. Chapter 3) is able to automatically provide such details to developers. Furthermore, it provides capabilities to adapt and adjust all algorithms for different languages and settings. As a result it provides explicit variability relations (i.e., whether elements are mandatory, alternative or optional) for different use cases to developers.

4.9. Chapter Summary

Previously, working with large sets of related variants (e.g., to identify and fix bugs) involved great effort as developers had to manually compare all existing variants with each other. Instead, developers can now apply Family Mining to their variants and easily search for parts containing an identified bug by analyzing a single 150% model. Using the described Family Mining algorithms from Section 4.4 (i.e., the generic EFA algorithm), Section 4.5 (i.e., the generic SBM algorithm) and Section 4.6 (i.e., a custom merge implementation for the used language following the presented ideas) it is now possible to execute custom-tailored Family Mining of new languages. Furthermore, additional algorithms (i.e., the MWA algorithm) from Section 4.7 allow to apply Family Mining even for cases, where hierarchy shifts would effectively prevent identification of sensible variability relations. By relying on meta-modeling techniques and customizable metrics, we are able to realize large parts of the Family Mining algorithms on a language-agnostic level and leave language-specific comparison logic to metrics specifically implemented for the applied language and current settings. Furthermore, by giving clear guidelines on how to adapt the algorithms and providing corresponding tooling (i.e., the generation facilities of the Vampire DSL), we allow easy adaptation of Family Mining for new block-based languages. Using the language-specific metrics, developers have full control over the executed comparisons between model entities and decide over the granularity of these comparisons and the impact of different attributes on the overall similarity. By using the merged 150% models (cf. Figure 4.10), the identified fine-grained variability information can be used by domain experts to execute a detailed analysis of the variability contained in the compared variants. For example, they are able to identify all model variants containing erroneous parts and apply implemented fixes to all corresponding model variants. Thus, the Family Mining approach is capable of reducing the maintenance effort for domain experts as it can provide such detailed variability information without tedious manual comparisons of relevant variants.

5 Migrating the Variability Information to Reusable Software Product Line Artifacts

The contents of this chapter are largely based on the work published in [WRS+17, SWS19].

Summary *Manually migrating a set of existing product variants to managed reuse in an SPL involves large effort for developers. Identified variability has to be encoded in reusable software artifacts. Features have to be identified across the variants to enable configuration based on corresponding feature models. To overcome these challenges, we describe our automatic approach to encode variability information identified by our* Family Mining *approach in a delta-oriented SPL. By analyzing the 150% models provided by the* Family Mining*, we are able to automatically derive features and corresponding delta modules implementing them. To allow configuration of products to be generated from the corresponding SPL, we also derive feature models with possible constraints to correctly encode their relations. Furthermore, we provide tooling to support developers in refactoring the automatically generated SPL towards their requirements.*

Migrating a set of product variants to an SPL requires upfront knowledge about their variability relations to correctly represent them in SPL artifacts allowing generation of all variants. Such detailed information about the variability relations between variants can be provided by our Family Mining approach in Chapter 4 in form of 150% models (cf. Definition 2.10). In this chapter, we describe the details for a variant migration to a delta-oriented SPL and assume that the 150% model is given in form of a meta-model instance conforming to Definition 4.3. While we realized our SPL migration based on the results of our Family Mining approach, it is also possible to use results from other automatic variability mining approaches or even manual analyses. The only prerequisite is that these approaches provide the necessary details and store them in the correct format.

Our approach uses delta modeling as the underlying variability realization mechanism for the created SPL, because delta modeling provides different advantages over other mechanisms (cf. Section 2.3.6). For example, it provides modularity (i.e., variability can be encapsulated in delta modules of different granularity), relies on language constructs easily understandable for domain experts (i.e., it uses a delta language specifically designed for the used modeling language) and, most importantly, allows high flexibility (i.e., reactive variant creation). Thus, delta modeling supports developers to understand the extracted SPL artifacts and allows companies to easily support new variants by adding new delta modules with additional functionality. For a concrete realization of our concepts, we use DeltaEcore (cf. Section 2.3.6) as it provides all necessary means to realize delta-oriented SPLs in model-based settings. For instance, it supports creation of delta languages for Ecore-based languages, feature modeling and variant generation using selected delta modules.

In Figure 5.1, we give an overview of our approach for migrating a set of product variants to the managed reuse in an SPL. Our MATADOR SPL[1] approach consists of four basic phases:

- *Delta Operation Identification:* This step analyzes the annotations from the input 150% model and derives for each contained model element the necessary delta operation with respect to a user-selected model (i.e., the model that is transformed by the generated delta operations) of the SPL. These delta operations form the basis for the actual transition to an SPL.
- *Delta Dialect Generation:* Based on the identified delta operations, this step generates a delta dialect for the variants encoded in the future SPL. This dialect allows automatic derivation of a corresponding delta language to transform instances of the used meta-model with matching language-specific delta operations.
- *Feature Identification:* This optional step analyzes user-provided information on the features contained in the migrated product variants. Based on this analysis, each model element is assigned to a feature and corresponding delta modules can be generated.
- *SPL Artifact Generation:* This step uses the generated delta dialect and comprises the *Delta Module Generation*, the *Feature Model Generation* and the *Mapping Generation*. These different generation steps encode the identified required delta operations in delta-oriented SPL artifacts implementing the variability between the input variants. The generated artifacts comprise delta modules with application orders, a feature model with corresponding configurations for the input variants, mappings between features and delta modules and recommendations for constraints between identified features.

[1] MIGRATING AUTOMATICALLY TOWARDS A DELTA-ORIENTED SOFTWARE PRODUCT LINE

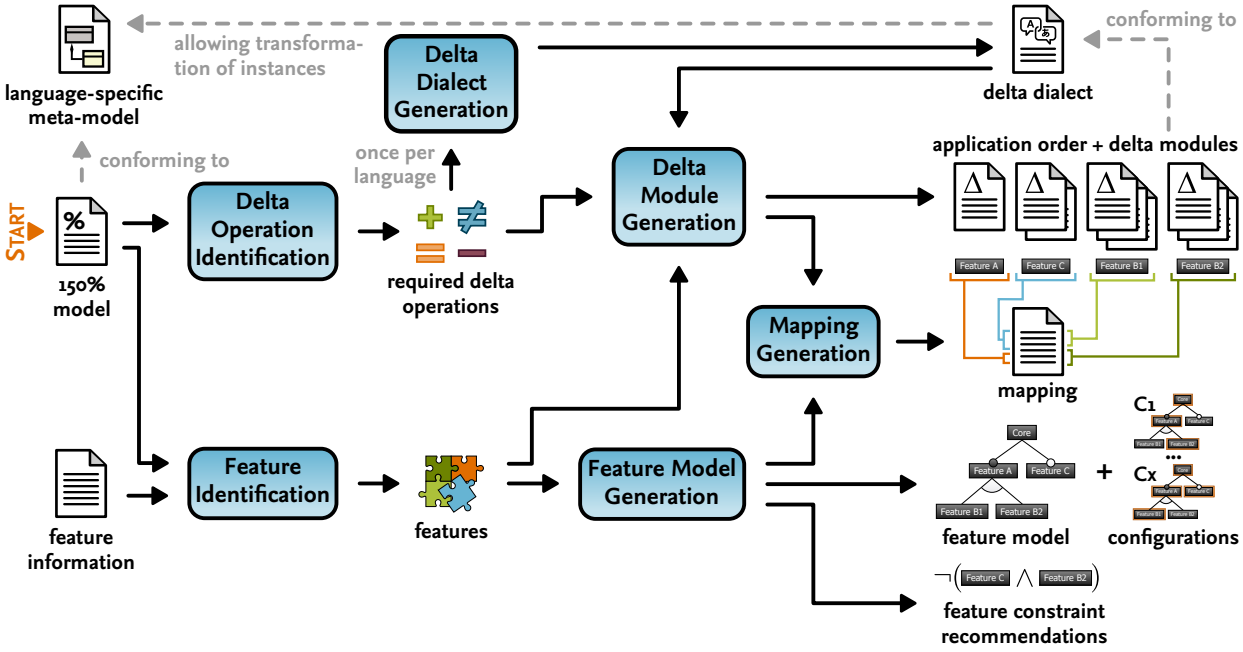


Figure 5.1.: Workflow of the MATADOR SPL approach to migrate to a delta-oriented software product line.

Chapter Outline The focus of the Matador SPL approach lies on its adaptability for different languages and the understandability for domain experts executing the transition to delta-oriented SPLs. Thus, we provide means to automatically derive delta languages for new modeling languages to lower the barrier for adopting managed reuse in SPLs. Furthermore, the generated SPL realization artifacts (i.e., the delta modules expressing the identified variability) are structured closely to the input implementations. This way, we enable traceability for developers between the original variants and the corresponding SPL realization. In addition, we allow for manual refactoring of the automatically generated artifacts towards an SPL conforming to the requirements of the user (e.g., company guidelines).

In the following sections, we explain the details of each step by applying the described techniques to our running example (cf. Section 2.2):

- *Section 5.1:* This section gives details of the algorithms to determine the required delta operations for each model element in the analyzed 150% model. The algorithm is used during the generation of delta dialects as well as delta modules.
- *Section 5.2:* Based on the identified delta operations, we describe how to generate a delta language for the modeling language used to implement the analyzed variants.
- *Section 5.3:* Using the identified necessary delta operations and the generated delta language, we are able to derive delta modules expressing the variability between the analyzed product variants and allowing their generation using DeltaEcore's facilities.
- *Section 5.4:* In case the users have additional knowledge of the input variants' implementation and their features, we can automatically derive a modular DeltaEcore SPL consisting of a feature model with delta modules storing the features' implementation details.
- *Section 5.5:* The resulting artifacts form the basis for an SPL realization that can be custom-tailored towards the requirements of users. We provide corresponding refactoring operations to support developers during such a restructuring and preserve the relations between artifacts.
- *Section 5.6:* This section provides a comparison with related work in the area of migrating product variants to an SPL, deriving variability languages and refactoring SPLs.

5.1. Determining the Required Delta Operations

During the migration of existing variants to an SPL, our Matador SPL approach requires information on the necessary delta operations to transform a user-selected variant into one of the other variants that are migrated to the created SPL. These delta operations have to specifically reflect the identified variability relations between the migrated variants to allow their derivation from the created SPL. The algorithm described in this section is only used to identify these required delta operations, but leaves the concrete interpretation of the information to the executed approach depending on the use case. On the one hand, the Matador SPL approach uses the identified information in Section 5.2 to generate the delta operation specifications of a delta language that supports the modeling language used for the migrated variants. On the other hand, the Matador SPL approach uses the identified information in Section 5.3 to generate delta modules encoding the variability between the migrated variants in form of delta operation calls to transform the user-selected variant.

We refer to the variant that is transformed by the identified delta operations as follows.

Definition 5.1: SPL Core Model Variant

The *SPL core model variant* mv_{core} (or SPL core for short) is a user-selected variant that serves as a basis for the delta-oriented SPL generated by the Matador SPL approach.

By allowing manual selection of the SPL core, we facilitate flexibility regarding the realization of the generated SPL. For instance, selecting a variant with artifacts common to the other variants allows generation of an SPL consisting mostly of additive variability (i.e., delta modules adding functionality). Whereas selecting a variant that has additional functionality over other variants might result in an SPL consisting of mostly negative variability (i.e., delta modules removing functionality). There is no definite answer on how to select the core variant for an SPL. Many different solutions are imaginable depending on the use case. However, we prefer to select a core that is mostly shared across all variants to reduce the number of delta operations generated to transform this variant into the other variants (i.e., adding additional functionality).

Based on the selected SPL core, the Matador SPL approach processes the input 150% model and determines for each contained model element, which operations have to be executed to transform this core into any of the other variants. We refer to these variants as follows.

Definition 5.2: Transformed Model Variants

A *transformed model variant* mv_{trans} represents a variant that is generated after applying the variant-specific delta operations identified by the Matador SPL approach to the SPL core variant mv_{core} (cf. Definition 5.1). All variants contained in a 150% model except for the user-selected SPL core are transformed variants.

For the identification of necessary delta operations, the approach fully relies on the 150% model annotations about the containment of model elements in the different variants (cf. Definition 4.3). Thus, the delta operation identification can only be as good as the variability details in the provided 150% model. If this information is of poor quality (e.g., elements are assigned to the wrong variants), the Matador SPL approach will fail and is not able to correctly derive the needed delta operations. Although we identified during evaluation of our Family Mining approach in academic and industrial settings that it provides reliable variability information, we allow manual adjustment of the 150% model prior to the SPL migration. In fact, we recommend an, at least, superficial analysis of the identified variability by domain experts to strengthen the understanding of the contained variability (especially in scenarios with unknown relations), but also to ensure correct results.

In Algorithm 5.1, we show the `identifyDeltaOperations` method triggering the identification of delta operations to transform the user-selected SPL core variant to another variant. During the delta operation identification, the Matador SPL approach relies on a *Breadth-First Search (BFS)*[2] of the input 150% model. To realize the traversal of 150% models in a language-independent way, the algorithm exploits the Ecore meta-model that serves as an underlying basis for each language-specific meta-model in the EMF. Thus, the short forms *eObj* and *eRef* in the following algorithms refer to the Ecore classes `EObject` and `EReference`, respectively. The algorithm expects as input

[2] The BFS algorithm was discovered by Moore [Moo59] when developing algorithms to efficiently find shortest paths in mazes. The same algorithm was independently developed by Lee [Lee61] for routing wires on circuit boards [CLR+09].

the identifiers id_{core} and id_{trans} used to annotate the SPL core variant mv_{core} and the currently processed transformed variant mv_{trans} in the analyzed 150% model. In addition, the `EObject` $eObj_{mr}$ for the 150% model root and the current hierarchy level h_c (i.e., $h_c = 0$ for the start of the delta operation identification) are needed by the algorithm. As an optional parameter, the method allows to pass the root feature f_{root} of a feature model that can be automatically generated during SPL generation. This can be used to assign the delta operations to features and to generate a modular SPL with corresponding delta modules comprising only a subset of all identified delta operations. The queue for the BFS approach is initialized with the delta operation determined by the `identifyDelta` method for $eObj_{mr}$ in Line 3.

Definition 5.3: Delta Operation δ_{op}

A determined delta operation δ_{op} stores the following information:

- id_{core} of the corresponding SPL core variant mv_{core}.
- id_{trans} of the transformed variant mv_{trans} for which the delta operation is calculated.
- The `EObject` $eObj_r$ whose delta operation is calculated.
- Parent `EReference` $eRef_p$ referencing $eObj_r$.
- Parent `EObject` $eObj_p$ that references $eObj_r$ via `EReference` $eRef_p$.
- Hierarchy level h of δ_{op} in the analyzed 150% model.
- Feature f the model element $eObj_r$ is belonging to (this information is optional).
- The actual delta operation δ that has to be executed for $eObj_r$ during a transformation of SPL core variant mv_{core} to transformed variant mv_{trans}.

Afterwards, the algorithm processes the remaining 150% model using a BFS strategy (cf. Line 4 – 8) until the complete model is traversed. By using this strategy, the algorithm ensures that all generated delta operations follow the order of the model hierarchy (i.e., a subsequent delta operation for an `EObject` is not processed prior to the parent `EObject` referencing it). While the queue still contains delta operations d^c_{op}, it adds them to the final list of all identified delta operations Δ_{op} for the current id_{trans} (cf. Line 4 – 8). Furthermore, the algorithm executes the `addReferenced-ElementsToQueue` method to identify `EObjects` referenced by the `EObject` of the current operation d^c_{op}, determines their corresponding delta operations and adds them to the queue (cf. Line 7).

In Algorithm 5.2, we show the `addReferencedElementsToQueue` method that processes the current delta operation δ^p_{op} from the queue to determine the delta operations for the `EObjects` referenced by its contained `EObject`. This method identifies all `EReferences` $ERef_p$ going out from $eObj_p$ contained in δ^p_{op} and further analyzes each of them (cf. Line 6 – 27). First, the algorithm checks whether the current `EReference` $eRef_p$ was not yet processed by another iteration (cf. Line 7). This check is necessary as each `EObject` contained in the processed 150% model might be referenced multiple times. As a result, identification of subsequent delta operations might otherwise be triggered multiple times. Afterwards, the algorithm retrieves the referenced `Object` obj_r (cf. Line 8) and checks whether it is an `EList` (cf. Line 9). In case *not*, it processes the single refer-

Input: identifier id_{core} for SPL core model variant mv_{core},
identifier id_{trans} for transformed model variant mv_{trans},
150% model root $eObj_{mr}$, feature model root f_{root},
current hierarchy level h_c
Output: all identified delta operations Δ_{op}

```
1  method identifyDeltaOperations(id_core, id_trans, eObj_mr, f_root, h_c) : Δ_op is
2      this.Q ← ∅
3      this.Q ← this.Q ⋃ identifyDelta(id_core, id_trans, ∅, ∅, ∅, eObj_mr, h_c, f_root)
4      while this.Q ≠ ∅ do
5          δ^c_op ← pollElement(this.Q)
6          Δ_op ← Δ_op ⋃ δ^c_op
7          addReferencedElementsToQueue(id_core, id_trans, δ^c_op)
8      end
9      return Δ_op
10 end
```

ll. 2 – 3: Initialization of the BFS queue. Uses the schema in Table 5.1.

ll. 5 – 6: Poll the current delta operation δ^c_{op} from queue **this**.Q and add it to the result Δ_{op}.

l. 7: Process the model elements referenced by the *eObj* in δ^c_{op}.

Algorithm 5.1: Main method of the delta operation identification in the Matador SPL approach.

enced `EObject` $eObj_r$ (cf. Line 11 – 15), otherwise it processes each of the `EObjects` $eObj_r$ stored in the corresponding `EList` $eList_r$ (cf. Line 18 – 24). For each processed $eObj_r$, the algorithm checks whether it stores the required variability information (i.e., its containing model variants) and identifies the corresponding delta operation δ^r_{op} by calling the `identifyDelta` method. In addition, the algorithm remembers for each $eObj_r$ that it was processed to prevent infinite loops.

In Table 5.1, we present the schema followed by the `identifyDelta` method (cf. ll. 11 – 15 & ll. 19 – 23) to determine the delta operation δ for a referenced `EObject` $eObj_r$ in relation to its parent $eObj_p$. The method derives the necessary operation by checking whether id_{core} and id_{trans} are contained in the model annotations ID_{mv} of $eObj_r$ and $eObj_p$, respectively. For instance, in case *14* $eObj_p$ is contained in mv_{core} ***and*** mv_{trans} (i.e., $id_{core} \in eObj_p.ID_{mv}$ and $id_{trans} \in eObj_p.ID_{mv}$). Thus, the method identifies that $eObj_r$ has to be added to $eRef_p$ as it is ***not*** contained in mv_{core} ***but*** in mv_{trans} (i.e., $id_{core} \notin eObj_r.ID_{mv}$ and $id_{trans} \in eObj_r.ID_{mv}$). In accordance to delta operations used by DeltaEcore the method distinguishes between adding or removing elements to / from a multi-valued or single-valued `EReference` $eRef_p$ (cf. Section 2.3.6). This allows the algorithm to correctly select the required DeltaEcore operation (e.g., δ_{add} vs. δ_{set} for additions). Furthermore, the method distinguishes cases where `EObject` $eObj_r$ is contained in both model variants and modifications were applied (i.e., δ_{modify}) and cases where the contained `EObject` $eObj_r$ can remain unchanged during a transformation (i.e., $\delta_{\emptyset}$). Please note that while cases *8* and *12* might seem unusual, they are necessary in situations where an element B is added to / removed from a reference in element A (i.e., through cases *14* and *15*, respectively) and element B holds a back reference to its parent element A (e.g., to indicate that element A is parent of B as for `Regions` in our statechart meta-model). For such situations, cases *8* and *12* are able to properly maintain this back reference by adding / removing the element correspondingly. All $\delta_{\emptyset}$ operations will be ignored in subsequent processing steps as they do not have any impact during variant derivation from the generated SPL.

Input: SPL core model identifier id_{core}, transformed model identifier id_{trans}, parent delta operation δ^p_{op} for the subsequent delta operations

```
1  method addReferencedElementsToQueue(id_core, id_trans, δ^p_op) : void is
2      eObj_p ← δ^p_op.eObj_r
3      h_p ← δ^p_op.h
4      f_p ← δ^p_op.f
5      ERef_p ← eObj_p.ERef
6      foreach eRef_p ∈ ERef_p do
7          if !referencePreviouslyProcessed(eObj_p, eRef_p) then
8              obj_r ← eRef_p.obj_r
9              if !isEList(obj_r) then
10                 eObj_r ← obj_r
11                 if storesVariability(eObj_r) then
12                     δ^r_op ← identifyDelta(id_core, id_trans, eObj_p, eRef_p, eObj_r, h_p + 1, f_p)
13                     this.Q ← this.Q ⋃ δ^r_op
14                     setProcessed(eObj_p, eRef_p, eObj_r)
15                 end
16             else
17                 eList_r ← obj_r
18                 foreach eObj_r ∈ eList_r do
19                     if storesVariability(eObj_r) then
20                         δ^r_op ← identifyDelta(id_core, id_trans, eObj_p, eRef_p, eObj_r, h_p + 1, f_p)
21                         this.Q ← this.Q ⋃ δ^r_op
22                         setProcessed(eObj_p, eRef_p, eObj_r)
23                     end
24                 end
25             end
26         end
27     end
28     return
29 end
```

ll. 2 – 4: Retrieve information relevant for the subsequent deltas.

l. 5: All references going out from $eObj_p$.

ll. 11 – 15 & ll. 19 – 23: Identify and store δ^r_{op} for the $eObj_r$ referenced by $eObj_p$ via $eRef_p$. Relies on the schema shown in Table 5.1.

ll. 18 – 24: For referenced lists $eList_r$ process all contained objects.

Algorithm 5.2: Populating the queue for the delta operation identification in the Matador SPL approach.

Case	$eObj_p$		$eObj_r$		Decision
	mv_{core}	mv_{trans}	mv_{core}	mv_{trans}	
1	0	0	0	0	$\delta_{\emptyset}$*
2	0	0	0	1	$\delta_{\emptyset}$*
3	0	0	1	0	$\delta_{\emptyset}$*
4	0	0	1	1	$\delta_{\emptyset}$*
5	0	1	0	0	$\delta_{\emptyset}$*
6	0	1	0	1	δ_{set} / δ_{add}
7	0	1	1	0	$\delta_{\emptyset}$*
8	0	1	1	1	δ_{set} / δ_{add}
9	1	0	0	0	$\delta_{\emptyset}$*
10	1	0	0	1	$\delta_{\emptyset}$*
11	1	0	1	0	δ_{unset} / δ_{remove}
12	1	0	1	1	δ_{unset} / δ_{remove}
13	1	1	0	0	$\delta_{\emptyset}$*
14	1	1	0	1	δ_{set} / δ_{add}
15	1	1	1	0	δ_{unset} / δ_{remove}
16	1	1	1	1	$\delta_{\emptyset}$ / δ_{modify}

*these combinations are *either* not possible (e.g., due to illegal parent-child relations) or *no* delta operation has to be executed

Table 5.1.: Schema used by the `identifyDelta` method to determine the delta operation required for `EObject` $eObj_r$ during a transformation of SPL core variant mv_{core} to transformed variant mv_{trans}. 0 and 1 show whether the corresponding element is contained in the corresponding mv or not.

In addition to the identification of the necessary delta operation, each of the generated δ_{op} objects is assigned to a feature (in case the user provided corresponding information). For now, we assume that such feature information is not available and leave corresponding explanations for later.

Using the described algorithm, our Matador SPL approach would derive the delta operations in Table 5.2 for the 150% model in Figure 4.10 of our running example in Figure 2.5. The table shows the parent `EObject` $eObj_p$, the parent `EReference` $eRef_p$, the referenced $eObj_r$ and the identified decision. Furthermore, we included the hierarchy level h of the delta operation in the model, which is later used to sort the delta operations in a correct application order. For space reasons, we do not include the $\delta_{\emptyset}$ decisions in this excerpt as they do not influence the generation of the SPL parts in subsequent steps. They would contain decisions for the statechart root element, the root region, the `cls_unlock` and `cls_lock` states and the `key_pos_unlock / cls_locked=false; pw_enabled=true;` transition. As we can see, the Matador SPL approach also generates decisions for model elements that are referenced by model elements with assigned δ_{remove} operations (e.g., an δ_{unset} operation is derived for the source and target states of

removed transitions). One could argue that these delta operations are unnecessary as the corresponding reference (e.g., to the source or target state) would be removed with its parent (e.g., the corresponding transition). As a consequence, delta modules generated based on these decisions contain additional "unnecessary" delta operation calls. However, we intentionally decided to generate these operations in order to make removals explicit rather than leaving their interpretation to the developers. As a result, the whole variant derivation process based on the generated delta operations is completely transparent to the developer.

Overall, using this approach allows us to identify the necessary delta operations in Section 5.2 to generate a delta language supporting the modeling language of the migrated variants and in Section 5.3 to generate the delta modules for the migration of the variants to the created SPL.

$eObj_p$	$eRef_p$	$eObj_r$	**Decision**	**Hierarchy** h
`cls_unlock`	`outgoingTransitions`	`key_pos_lock [pw_pos != 1] / cls_locked=true;`	δ_{remove}	3
`cls_unlock`	`outgoingTransitions`	`key_pos_lock [pw_pos == 1] / cls_locked=true; pw_enabled=false;`	δ_{remove}	3
`cls_unlock`	`outgoingTransitions`	`key_pos_lock / cls_locked=true; pw_enabled=false; GEN(pw_but_up);`	δ_{add}	3
`cls_lock`	`incomingTransitions`	`key_pos_lock [pw_pos != 1] / cls_locked=true;`	δ_{remove}	3
`cls_lock`	`incomingTransitions`	`key_pos_lock [pw_pos == 1] / cls_locked=true; pw_enabled=false;`	δ_{remove}	3
`cls_lock`	`incomingTransitions`	`key_pos_lock / cls_locked=true; pw_enabled=false; GEN(pw_but_up);`	δ_{add}	3
`key_pos_lock [pw_pos != 1] / cls_locked=true;`	`source`	`cls_unlock`	δ_{unset}	4
`key_pos_lock [pw_pos != 1] / cls_locked=true;`	`target`	`cls_lock`	δ_{unset}	4
`key_pos_lock [pw_pos != 1] / cls_locked=true;`	`label`	*`TransitionLabel`*	δ_{unset}	4
`key_pos_lock [pw_pos == 1] / cls_locked=true; pw_enabled=false;`	`source`	`cls_unlock`	δ_{unset}	4
`key_pos_lock [pw_pos == 1] / cls_locked=true; pw_enabled=false;`	`target`	`cls_lock`	δ_{unset}	4
`key_pos_lock [pw_pos == 1] / cls_locked=true; pw_enabled=false;`	`label`	*`TransitionLabel`*	δ_{unset}	4
`key_pos_lock / cls_locked=true; pw_enabled=false; GEN(pw_but_up);`	`source`	`cls_unlock`	δ_{set}	4
`key_pos_lock / cls_locked=true; pw_enabled=false; GEN(pw_but_up);`	`target`	`cls_lock`	δ_{set}	4
`key_pos_lock / cls_locked=true; pw_enabled=false; GEN(pw_but_up);`	`label`	*`TransitionLabel`*	δ_{set}	4

Table 5.2.: Excerpt from the delta operations identified by our Matador SPL approach based on the 150% model in Figure 4.10 of the Central Locking System (CLS) variants from our running example in Figure 2.5. The hierarchy h is later used to sort the operations in a correct application order.

5.2. Generating Delta Languages for Used Modeling Languages

Based on the delta operations Δ_{op} identified by Algorithm 5.1 in Section 5.1, the Matador SPL approach can now trigger the generation of a delta language specifically tailored towards the used modeling language. Although DeltaEcore already provides the possibility to generate a delta language δ_{lang} for a given meta-model, our Matador SPL approach allows to generate delta languages $\delta_{lang}^{spl} \subset \delta_{lang}$. These δ_{lang}^{spl} comprise exactly the subset of delta operations from δ_{lang} that are needed to realize the generated SPLs. One can argue whether it is sensible to only support a subset of all delta operations available for a given modeling language. However, we decided to provide this additional capability as delta operations can be very invasive (e.g., complete model parts can be removed unintentionally) and allow to easily apply unwanted changes to model functionality (e.g., needed transitions might be removed). Thus, our approach allows to restrain the generated delta language to a subset of delta operations that is actually needed for the initial SPL realization. This way, unnecessary and potential "illegal" delta operations that might not be wanted by a company (e.g., due to design guidelines) are not provided to the developer. Furthermore, developers using the migrated SPL are not overwhelmed with delta operations that are unnecessary for the corresponding implementation. However, the restrained delta language δ_{lang}^{spl} can be manually extended with further delta operations or replaced with the complete δ_{lang} during restructuring and extension of the created SPL at any time if desired. Depending on the selected strategy, the delta language generation has to be done only once per modeling language when creating the complete delta language using the support of DeltaEcore or once per family of migrated model variants when using the Matador SPL approach to generate the restrained delta language δ_{lang}^{spl}. In this case, other families of model variants using the same modeling language might require other delta operations, which might not be included in the restrained delta language δ_{lang}^{spl}.

For each delta operation derived during the previous identification, the Matador SPL approach first identifies which type of delta operation from the common base delta language in DeltaEcore is needed to implement the operation. It then generates a corresponding delta operation specification in the created delta dialect with the necessary arguments and an automatically generated operation name.

Definition 5.4: Delta Operation Signature

The *signature* for a delta operation δ_{op} differs from the signature of methods in classic programming languages and consists of:

- A *type* showing the executed delta operation (e.g., a remove operation).
- A unique operation *name* for the used delta dialect.
- A set of *parameters* indicating which *values* for which *references* of specific *meta-model classes* are changed.

Thus, a delta operation δ_{op} has no return type, but an operation type showing its functionality.

In Listing 5.1, we show a delta dialect generated for the 150% model in Figure 4.10 for our running example in Figure 2.5. This delta dialect references the *Uniform Resource Identifier (URI)* of our simplified statechart meta-model in Figure 4.3 (cf. Line 4). The names of the delta operations are generated in a way that they are human-understandable. For example, the delta operation to set the label of a transition concatenates the executed action (i.e., `set`) with the name of the `EReference` $eRef_p$ holding the label (i.e., `label`) and the name of `EObject` $eObj_p$ that holds $eRef_p$ (i.e., `Transition`). These generated operation names can be edited by developers after generating the dialect as long as the signature remains unchanged, because the subsequent delta module generation relies on it. For instance, a developer might prefer a shorter name for the `setLabelOfTransition` delta operation and to use `setLabel` instead. In case the names of the methods are changed, developers have to keep in mind that any delta modules that were generated or manually created using corresponding delta operation calls might not find the renamed operation.

The signature of methods is created similar to the operation names. First, the Matador SPL approach adds the `EObject` $eObj_r$ as the value for the method if necessary (i.e., for δ_{unset} operations only information on the cleared reference is needed). Afterwards, the approach adds the `EObject` $eObj_p$ with the changed `EReference` $eRef_p$ in brackets. For example, the signature of the delta operation to set the label of a transition holds the `TransitionLabel` as the first parameter and the `Transition's label` reference as a second parameter. By maintaining a set of all delta operations generated for a delta dialect, we prevent generation of redundant operations in cases where an operation is called multiple times.

During the execution of δ_{add} or δ_{set} operations, it is sometimes necessary to initialize `EAttribute` lists for newly created and added model elements. For example, the `TransitionLabel` class in our simplified statechart meta-model in Figure 4.3 uses `EAttribute` lists of `Strings` to store `events`, `conditions` and `actions` of a `Transition`. However, DeltaEcore only provides means to initialize single-valued `EAttributes`, and we are not able to initialize multi-value `EAttributes` during element creation nor to use one of the base delta operations to add elements to such `EAttribute` lists. To overcome this limitation, the Matador SPL approach additionally generates corresponding custom delta operations to allow initialization of such lists. Normally, the executed functionality of such custom delta operations has to be manually implemented. However, for our Matador SPL approach, we provide further generation facilities to automatically derive the corresponding custom implementations.

In Listing 5.2, we show an example for the code generated by our Matador SPL approach for adding a `String` to a `TransitionLabel`. Based on the `attributeName` of the list (e.g., `events`) in the `EObject` $eObj_r$ (e.g., `TransitionLabel`) that is created during execution of δ_{add} or δ_{set} operations, our approach searches for the corresponding `EAttribute` and adds the given `String` value to the identified list.

Using the automatically generated delta dialect, it is possible to derive a corresponding delta language by executing the corresponding facilities provided by DeltaEcore (cf. Section 2.3.6).

```
deltaDialect {
  configuration:
    metaModel:
      <http://www.tu-bs.de/isf/familymining/stateoriented>;

  deltaOperations:
    addOperation
      addTransitionToIncomingTransitionsOfAbstractState(
        Transition value,
        AbstractState [incomingTransitions] element);
    removeOperation
      removeTransitionFromIncomingTransitionsOfAbstractState(
        Transition value,
        AbstractState [incomingTransitions] element);
    addOperation
      addTransitionToOutgoingTransitionsOfState(
        Transition value,
        State [outgoingTransitions] element);
    removeOperation
      removeTransitionFromOutgoingTransitionsOfState(
        Transition value,
        State [outgoingTransitions] element);
    setOperation setSourceOfTransition(AbstractState value,
      Transition [source] element);
    unsetOperation unsetSourceOfTransition(
      Transition [source] element);
    setOperation setTargetOfTransition(AbstractState value,
      Transition [target] element);
    unsetOperation unsetTargetOfTransition(
      Transition [target] element);
    setOperation setLabelOfTransition(TransitionLabel value,
      Transition [label] element);
    unsetOperation unsetLabelOfTransition(
      Transition [label] element);

    customOperation addStringToStringsListOfTransitionLabel(
      String value, TransitionLabel element,
      String attributeName);
}
```

Listing 5.1: DeltaEcore delta dialect generated by the Matador SPL approach from the delta operations in Table 5.2 for the Central Locking System (CLS) variants in Figure 2.5.

```
1  @Override
2  protected boolean interpretAddStringToStringsListOfTransitionLabel(
3      DEModelWriter modelWriter, String value,
4      TransitionLabel element, String attributeName)
5  {
6    EAttribute foundEAttribute = null;
7    EClass elementEClass = element.eClass();
8    for (EAttribute attribute : elementEClass.getEAllAttributes()) {
9      if (attribute.getName().equals(attributeName)) {
10       foundEAttribute = attribute;
11       break;
12     }
13   }
14
15   if (foundEAttribute != null) {
16     Object currentEAttributeValue = element.eGet(foundEAttribute);
17
18     if (currentEAttributeValue instanceof EList<?>) {
19       EList<String> currentValueList =
20                 (EList<String>) currentEAttributeValue;
21       currentValueList.add(value);
22     }
23   }
24
25   return true;
26 }
```

Listing 5.2: Code generated by the Matador SPL approach to initialize a list of `String` values in the `TransitionLabel` class of our simplified statechart meta-model in Figure 4.3.

5.3. General Approach for the Generation of Delta Modules

Using the delta operations Δ_{op} identified by Algorithm 5.1 in Section 5.1 and a generated delta language (cf. Section 5.2) or a manually created delta language, it is now possible to encode the identified variability between the variants of the analyzed 150% model in delta modules for the created SPL.

Before actually encoding the identified variability between the SPL core mv_{core} and a transformed variant mv_{trans} in delta modules, the Matador SPL approach sorts the delta operations Δ_{op} to ensure their correct execution order. For instance, when adding states to the region of a statechart, the corresponding parent region has to exist. Otherwise, the execution of the δ_{add} delta operation for the states will fail. Similar, when removing a region, we first have to remove its contents. Matador SPL ensures the following execution order:

$$\forall \delta_{op} \in \Delta_{op} \text{ holds } \{\delta_{unset}, \delta_{remove}\} < \{\delta_{set}, \delta_{add}\} < \delta_{modify}$$

In addition, Matador SPL sorts the set of $\{\delta_{unset}, \delta_{remove}\}$ operations using the following rules:

1. *bottom-up execution of removals:* Elements that are deeper in the model hierarchy have to be removed first. Thus, we start execution of δ_{unset} or δ_{remove} operations that have the highest hierarchy level h and ensure for δ_{unset} and δ_{remove} operations:

$$\forall \{\delta_{unset}, \delta_{remove}\} \in \Delta_{op} \text{ holds } \delta_{op}.h_n < \delta_{op}.h_{n-1} < \ldots < \delta_{op}.h_0$$

2. *prioritize non-containment references:* References to objects have to be removed prior to removing the actual object. Thus, δ_{unset} or δ_{remove} operations with the same hierarchy level h are sorted according to the containment of their `EReference`. As a result, δ_{unset} or δ_{remove} operations on non-containment references $eRef_p^{nc}$ are executed *prior* to containment references $eRef_p^c$:

$$\forall \{\delta_{unset}, \delta_{remove}\} \in \Delta_{op} \text{ where } \delta_{op_1}.h \equiv \delta_{op_2}.h \text{ holds } \delta_{op}.eRef_p^{nc} < \delta_{op}.eRef_p^c$$

Similarly, the set of $\{\delta_{set}, \delta_{add}\}$ operations is sorted according to the following rules:

1. *bottom-up execution of additions:* Elements that are higher in the model hierarchy have to be added first to allow addition of further sub elements. Thus, we start execution of δ_{set} or δ_{add} operations that have the lowest hierarchy level h and ensure for δ_{set} and δ_{add} operations:

$$\forall \{\delta_{set}, \delta_{add}\} \in \Delta_{op} \text{ holds } \delta_{op}.h_0 < \delta_{op}.h_1 < \ldots < \delta_{op}.h_n$$

2. *prioritize containment references:* Objects have to be added to the model prior to adding references to the object. Thus, δ_{set} or δ_{add} operations with the same hierarchy level h are sorted according to the containment of their `EReference`. As a result, δ_{set} or δ_{add} operations on containment references $eRef_p^c$ are executed *prior* to non-containment references $eRef_p^{nc}$:

$$\forall \{\delta_{set}, \delta_{add}\} \in \Delta_{op} \text{ where } \delta_{op_1}.h \equiv \delta_{op_2}.h \text{ holds } \delta_{op}.eRef_p^c < \delta_{op}.eRef_p^{nc}$$

Based on the sorted delta operations, the Matador SPL approach executes for the delta operations Δ_{op} of each transformed variant mv_{trans} in the analyzed input 150% model a look-up process.

This process determines for each δ_{op} the necessary operation from the used delta language that will be called in the generated delta module. Starting with all possible operations of the dialect, the look-up process:

1. Reduces the possible operations from the dialect to a set of sensible operations based on the required operation type of δ_{op}. For example, for a δ_{add} only add operations from the delta dialect are considered for the subsequent look-up process.
2. Identifies an operation with a matching signature based on the required signature of δ_{op}. For instance, for an $eObj_r$ of type `TransitionLabel` that should be set to an $eRef_p$ `label` of an $eObj_p$ `Transition`, the process would rule out the `setSourceOfTransition` and `setTargetOfTransition` operations, because of their inappropriate signature and select the `setLabelOfTransition` operation.

The identified operation is used to create a call with the corresponding values as parameters, which is added to the generated delta module. Furthermore, in case of δ_{set} and δ_{add} operations, the approach adds constructor calls and, if necessary, calls to the custom list initialization operations to initialize objects that were not previously contained in the core model.

In Listing 5.3, we show the delta module generated for the delta operations in Table 5.2 for the Central Locking System (CLS) variants from our running example in Figure 2.5. First, in Line 1 a name for the delta module is assigned based on the name of the SPL core mv_{core} and transformed variant mv_{trans}. Furthermore, Line 2 references the used delta dialect. In addition, the `modifies` keyword in Line 3 references the SPL core. In this example, we can also see the construction of previously not included model objects (cf. Line 27 and Line 37) and an initialization of attribute lists using a generated custom operation (cf. Line 38 – 45).

```
configuration delta "ManPW -> AutoPW"
  dialect <http://www.tu-bs.de/isf/familymining/stateoriented>
  modifies <ManPW.statechart>
{
  unsetLabelOfTransition(<key_pos_lock [pw_pos != 1] /
    cls_locked=true;>);
  unsetLabelOfTransition(<key_pos_lock [pw_pos == 1] /
    cls_locked=true; pw_enabled=false;>);
  unsetSourceOfTransition(<key_pos_lock [pw_pos != 1] /
    cls_locked=true;>);
  unsetTargetOfTransition(<key_pos_lock [pw_pos != 1] /
    cls_locked=true;>);
  unsetSourceOfTransition(<key_pos_lock [pw_pos == 1] /
    cls_locked=true; pw_enabled=false;>);
  unsetTargetOfTransition(<key_pos_lock [pw_pos == 1] /
    cls_locked=true; pw_enabled=false;>);
  removeTransitionFromOutgoingTransitionsOfState(
    <key_pos_lock [pw_pos != 1] / cls_locked=true;>, <cls_unlock>);
  removeTransitionFromOutgoingTransitionsOfState(
    <key_pos_lock [pw_pos == 1] / cls_locked=true;
      pw_enabled=false;>, <cls_unlock>);
  removeTransitionFromIncomingTransitionsOfAbstractState(
    <key_pos_lock [pw_pos != 1] / cls_locked=true;>, <cls_lock>);
  removeTransitionFromIncomingTransitionsOfAbstractState(
    <key_pos_lock [pw_pos == 1] / cls_locked=true;
      pw_enabled=false;>, <cls_lock>);
  Transition transition = new Transition(id: "key_pos_lock /
    cls_locked=true; pw_enabled=false; GEN(pw_but_up);");
  addTransitionToOutgoingTransitionsOfState(transition,
    <cls_unlock>);
  addTransitionToIncomingTransitionsOfState(transition,
    <cls_lock>);
  setSourceOfTransition(<cls_unlock>, <key_pos_lock /
    cls_locked=true; pw_enabled=false; GEN(pw_but_up);>);
  setTargetOfTransition(<cls_lock>, <key_pos_lock /
    cls_locked=true; pw_enabled=false; GEN(pw_but_up);>);
  TransitionLabel label = new TransitionLabel(id: "label");
  addStringToStringsListOfTransitionLabel("key_pos_lock",
    label, "events");
  addStringToStringsListOfTransitionLabel("cls_locked=true;",
    label, "actions");
  addStringToStringsListOfTransitionLabel("pw_enabled=false;",
    label, "actions");
  addStringToStringsListOfTransitionLabel("GEN(pw_but_up);",
    label, "actions");
  setLabelOfTransition(label, <key_pos_lock / cls_locked=true;
    pw_enabled=false; GEN(pw_but_up);>);
}
```

Listing 5.3: DeltaEcore delta module generated by the Matador SPL approach based on the DeltaEcore delta dialect in Listing 5.1 to store the delta operations in Table 5.2 identified for the Central Locking System (CLS) variants from our running example in Figure 2.5.

5.4. Exploiting Information on the Features to Generate Feature Modules

To allow for a higher modularization of the generated SPL, our Matador SPL approach allows users to incorporate their knowledge of the variant implementations. This way, the approach is able to identify delta operation subsets $\Delta_{op}^{f} \subset \Delta_{op}$ comprising the functionality of product features.

5.4.1. Identifying Features and Generating Feature Delta Modules

After talking to domain experts and analyzing different academic and industrial Matlab/Simulink and statechart implementations, we came to the conclusion that a sensible way of searching for features is an analysis of the hierarchical decomposition of the implemented functionality. As mentioned in Section 2.1, hierarchical decomposition in model-based languages is a common means to start a complex implementation on an abstract level (e.g., with a basic architecture) and to refine its functionality with each added hierarchy level (e.g., by adding different features to process the input data). Thus, in model-based languages developers use different hierarchical containers, such as subsystems in Matlab/Simulink models and regions in statecharts, to implement coherent functionality and name them accordingly. In many cases this coherent functionality encapsulated in hierarchical containers can be referred to as a feature with respect to Definition 2.8. For example, in one of the analyzed Matlab/Simulink case studies from the automotive industry, we found a number of subsystems that encapsulated complete driver assistance system features, such as cruise control and emergency break, with corresponding names.

Automatic and Semi-Automatic Feature Identification Our approach to identify such features follows a rather simple and heuristic approach by traversing the complete model hierarchy and considering the names of hierarchical containers as identifiers for features. By providing a list of known feature names, the user of our Matador SPL approach is able to provide enough knowledge to identify the corresponding containers comprising features. We either allow for a completely automatic approach or allow users to semi-automatically select features from a list of potential candidates.

In Algorithm 5.3, we depict the algorithm for the `identifyFeatureObjects` method that is recursively called to traverse the 150% model in a DFS manner and to automatically identify features. Internally, we represent each feature by a so-called *feature object* storing all relevant information, such as its position in the model, its realizing delta operations and the containing variants. The method is triggered with a mandatory feature object f_{root} representing the root of the later generated feature model and the 150% model root `EObject` $eObj_{mr}$. The algorithm identifies all `EObjects` that are contained below the current $eObj_p$ and checks for each of them whether they represent the root model element of a feature by calling the `representsFeatureRoot` method (cf. Line 3). This abstract method can be extended by the user to execute the actual checks. For instance, in case of statecharts, we first check whether the current *eObj* represents an instance of the `Region` class from our statechart meta-model (cf. the simplified meta-model in Figure 4.3). In case this prerequisite is true, we check whether the name of the region is contained in the user-defined list of known feature names. Similarly, we check blocks in Matlab/Simulink models to be of type *Subsystem* and, only afterwards, check their names. In case, the algorithm identified a feature object root based on this assessment, we retrieve all relevant information from the corresponding model element *eObj* in the 150% model (cf. Line 4 – 6). This information includes the identifiers ID_{mv}^{f} of

Input: the parent feature object f_p, the parent model element $eObj_p$,
initial inputs are root feature f_{root} of the feature model and 150% model root $eObj_{mr}$

```
1  method identifyFeatureObjects(f_p, eObj_p) : void is
2      foreach eObj ∈ eObj_p.EObj do
3          if representsFeatureRoot(eObj) then
4              ID^f_mv ← identifyContainingModelVariants(eObj)
5              v_f ← identifyObjectVariability(eObj)
6              name_f ← identifyFeatureName(eObj)
7              f_n ← createFeatureObject(f_p, eObj, name_f, v_f, ID^f_mv)
8              identifyFeatureObjects(f_n, eObj)
9          else
10             identifyFeatureObjects(f_p, eObj)
11         end
12     end
13     return
14 end
```

ll. 2 – 12: Process all sub contents of $eObj_p$.

ll. 4 – 6: Retrieve relevant feature information from the $eObj$.

ll. 8 & 10: Recursively identify sub features.

Algorithm 5.3: Automatic identification of features in the Matador SPL approach.

all model variants containing the $eObj$, the variability v_f for $eObj$ and the $name_f$ of the $eObj$. Based on this information, we create a new feature object that is contained in the identified model variants and uses the derived variability and name. This newly created feature object serves as parent feature for any features located in the model hierarchy below $eObj$. After the overall feature identification, these parent-child relationships between the created feature objects directly show the hierarchical decomposition of the later generated feature model.

The semi-automatic identification of potential feature objects is realized in a similar manner. However, here, the user-extended `representsFeatureRoot` method only has to check, whether the currently analyzed model element represents a feature root (i.e., in case of statecharts, an instance of the `Region` class from our simplified statechart meta-model in Figure 4.3). All resulting feature objects are presented to the user in a tree structure allowing to manually analyze the candidate features and select the preferred features.

Generation of Delta Modules for the Identified Features For each variant containing one of the identified features f, the Matador SPL approach identifies all $\Delta_{op}^{f} \subset \Delta_{op}$ that are necessary to transform the corresponding variant accordingly. Depending on the selected SPL core mv_{core}, the feature either might need to be added or removed. Thus, although a specific variant might not contain feature f, it might be necessary to generate a corresponding delta module. For instance when selecting an SPL core mv_{core} of the *BCS* variants that contains the `ManPW` feature, the Matador SPL has to generate a delta module implementing the removal of the `ManPW` feature to allow generation of variants containing the alternative `AutoPW` feature.

To identify delta operations for features, the Matador SPL approach relies on the same algorithm as for the identification of delta operations for single delta modules per variant (cf. Algorithm 5.1 in Section 5.1). However, for the generation of delta modules per feature, this algorithm uses in-

formation on the identified features during the population of the delta operation δ_{op} queue in Algorithm 5.2. Each newly created delta operation δ_{op} is assigned to its parent feature f_p in method `identifyDelta`, unless the corresponding model element was identified as a root of a child feature f_c of f_p. In this case, the new delta operation δ_{op} for the current $eObj_r$ is assigned to f_c.

Thus, after executing this algorithm, the Matador SPL approach directly identified all necessary delta operations for each feature. Based on this information, the Matador SPL approach generates for each feature f the corresponding delta modules storing the necessary delta operations $\Delta_{op}^{f} \subset \Delta_{op}$. In Algorithm 5.4, we present the corresponding algorithm executed by the `generateDeltaModulesPerFeature` method. For each identified feature this method is called to derive corresponding delta modules. However, as the implementation of features in the different variants of the 150% model might be exactly the same, the algorithm does not simply create corresponding delta modules for each variant. Instead the algorithm identifies such equal implementations by traversing all delta operations identified for the feature in the variants stored in the input 150% model (cf. Line 2 – 18). First, the algorithm stores that the current delta module will contain the feature implementation for this variant (cf. Line 7) and retrieves the delta operations Δ_{op}^{fr} for the first variant id_{trans} as reference implementation (cf. Line 8). For each subsequent model variant stored in the 150% model, the algorithm checks whether the delta operations Δ_{op}^{ftrans} for the feature f are the same as for the reference feature implementation (cf. Line 9). In this case, the algorithm stores that the current delta module will store the feature implementation for this variant id_{trans} as well (cf. Line 10). After traversing all model variants from the 150% model, the algorithm generates and stores delta module dm (cf. Line 14 – 15) if the set of reference delta operations Δ_{op}^{r} is not empty (cf. Line 13). This can happen in cases where the feature is only realized by adding its functionality (i.e., non of the variants needs removal delta operations) and the current model variants in ID_{dm} do not implement the feature at all (i.e., it does not to be added and Δ_{op}^{fr} is empty). Afterwards, the model variants ID_{dm}, whose implementation of feature f is stored in delta module dm, are removed from all identifiers ID_{trans}. The algorithm continues this generation until all differing feature implementations are stored in corresponding delta modules.

Based on the generated delta modules, the Matador SPL approach might modify the tree structure of the identified feature objects to allow correct configurations. In case, differing implementations for a feature f were identified, the Matador SPL approach introduces additional child features f_a storing the alternative implementations of f. Otherwise, the differing feature implementation would not be selectable from the later generated feature model.

5.4.2. Generating Feature Models and Product Configurations

Based on the identified feature objects, we are able to generate a corresponding feature model with product configurations for the variants stored in the 150% model.

Feature Model Generation The feature model generation relies on the hierarchical structure (i.e., the parent-child relationships) of the identified feature objects and, thus, no complex generation algorithm is necessary. The corresponding generation starts from the mandatory feature object f_{root} introduced as a starting point for the feature identification (cf. Section 5.4.1) and traverses the tree structure of identified sub feature objects. During this traversal, we add child features to the feature model corresponding to the attributes identified for the feature objects (i.e., a feature is created with the corresponding name and the identified variability).

Input: current feature f, identifiers ID_{trans} of all transformed model variants in the 150% model
Output: all generated delta modules DM_f for feature f

```
1  method generateDeltaModulesPerFeature(f_c, ID_trans) : DM_f is
2      while ID_trans ≠ ∅ do
3          ID_dm ← ∅
4          Δ_op^fr ← ∅
5          foreach id_trans ∈ ID_trans do
6              if Δ_op^fr ≡ ∅ then
7                  ID_dm ← ID_dm ⋃ id_trans
8                  Δ_op^fr ← Δ_op^ftrans
9              else if variantsNeedSameDeltaOperations(Δ_op^fr, Δ_op^ftrans) then
10                 ID_dm ← ID_dm ⋃ id_trans
11             end
12         end
13         if Δ_op^fr ≠ ∅ then
14             dm ← createDeltaModule(ID_dm, f, Δ_op^fr)
15             DM_f ← DM_f ⋃ dm
16         end
17         ID_trans ← ID_trans \ ID_dm
18     end
19     return DM_f
20 end
```

l. 8: Delta operations of current variant id_{trans} will be stored in this delta module.

ll. 7 & 10: Feature f for variant id_{trans} will be represented by the generated delta module.

l. 17: Remove all variants ID_{dm} represented by delta module dm from ID_{trans}.

Algorithm 5.4: Generation of delta modules per feature in the Matador SPL approach.

In Figure 5.2, we present an exemplary DeltaEcore feature model generated based on automatically identified features for model variants from our *BCS* running example in Section 2.2. The generated SPL contains a large number of features and the Matador SPL approach was able to identify different implementations for the same feature (i.e., the `FP` and `CLS` features). Furthermore, we can see that the heuristic feature identification identified for the `HMI`, `LED`, `ManPW`, `AutoPW`, `FP`, `EM_heat`, `AS`, `EM`, `CLS`, `RCK` features (from left to right) additional parent features with the prefix `BCS_`. This is due to the fact, that for each of these feature the *BCS* implementation contains a region (i.e., the features *with* the `BCS_` prefix) with a hierarchical state containing the actual implementation (i.e., the features *without* the `BCS_` prefix). As both, the regions with and without the `BCS_` prefix, contain the user-specified feature name as a substring, they were reported back as identified features. Similarly, the `LED_AS` feature has three sub features that were identified as sub regions used to realize the functionality of the `LED_AS` feature. Furthermore, we can see that our Family Mining approach identified two `HMI` features (i.e., `HMI` and `HMI_Controller`). When looking at the actual implementation of these two features, we can see that they are realized exactly the same. However, in one case the developer decided to rename the feature for some reason. As a result, our merging algorithm decided to split the corresponding comparison element into two optional

regions as their names are not similar enough according to the Levenshtein Distance algorithm (cf. Section 4.6.2). While this shows the limitations of identifying features solely relying on the structure of models and the names of model elements, such related features can easily be merged in a subsequent refactoring of the generated SPL (cf. Section 5.5). Obviously, this involves manual restructuring of the generated artifacts. However, we argue that a completely automatic migration to an SPL is unrealistic anyway as different (e.g., company-specific) requirements would always require manual adjustment. In fact, we believe that such manual refactorings are eased by the close coupling between the generated SPL artifacts and the input implementations as developers can easily understand the relations between both realizations. Furthermore, we provide corresponding tooling to actually support developers during the refactoring of the generated SPL (cf. Section 5.5). Thus, we see this as a major advantage over other SPL migration techniques.

When comparing the feature model generated by our Matador SPL approach (cf. Figure 5.2) with the original *BCS* feature model (cf. Figure 2.9), we can see that differences exist. As expected the generated feature model is very close to the hierarchy of the corresponding statechart implementations for the analyzed variants of the *BCS*. Thus, the hierarchical decomposition of the feature model differs as the developers for the original *BCS* feature model (cf. Figure 2.9) used their domain knowledge to logically group related features by introducing additional features without concrete functionality (i.e., without corresponding model elements in the statechart implementation of the *BCS*). Examples are the `Door System` feature and the `Security` feature in Figure 2.9, which are both used to group corresponding sub features. Furthermore, not all features from the original *BCS* feature model are contained in the generated feature model as the model variants analyzed by the Matador SPL only comprised a subset of all possible features. Despite the differences in the identified feature names and the hierarchical decomposition, our feature identification was able to identify features that represent core functionality of the *BCS*. In addition, the Matador SPL approach was able to identify the correct variability for almost all features except the `ManPW` and `AutoPW` features. Here, the corresponding alternative constraint from the original *BCS* feature model is identified during further analysis of their relations to derive cross-tree constraints (cf. Section 5.4.3). Altogether, even without deep knowledge of the *BCS*, one can see that the identified features represent sensible configuration options and are close to the original feature model.

Prior to the creation of the feature model, we allow execution of additional algorithms to optimize optional features and to reduce the variability space of the created SPL. In case a parent feature f_p is contained in exactly the same variants as an optional child feature f_o of f_p, we change the variability of f_o to mandatory as, for the analyzed variants, these features are always used together. For instance, in Figure 5.2, all optional features below the features with a `BCS_` prefix or the additional `LED_AS` features below the parent `LED_AS` would be changed to mandatory features. This modification of the identified variability can be executed without loss of generality (i.e., incorrectly restricting the SPL's variability) as we only exploit knowledge from the existing variants. That is, the corresponding sub feature f_o is *always* used in conjunction with its parent feature f_p. This option allows to migrate to an SPL that is as close as possible to the original implementation of the input variants. However, as users might also want to exploit the newly identified variability to create new variants, this option is deactivated by default.

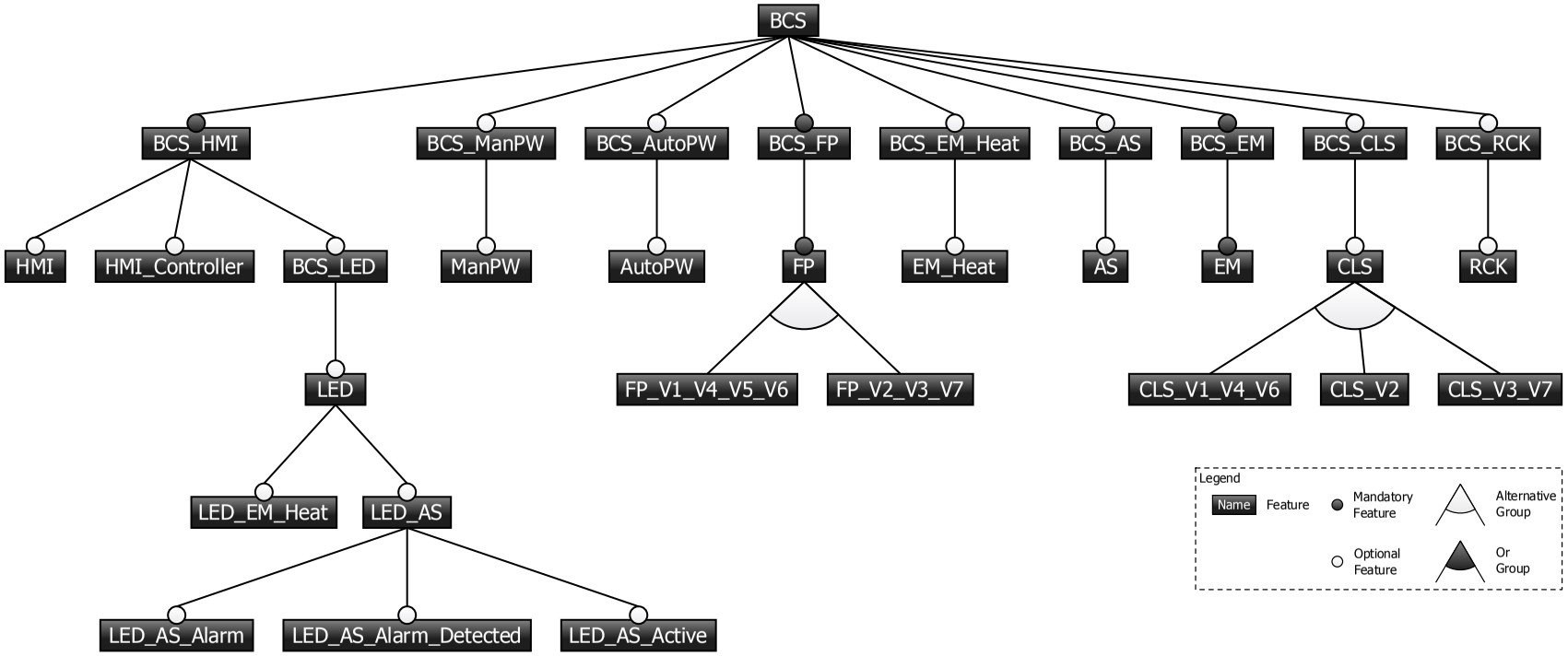


Figure 5.2.: Exemplary DeltaEcore feature model generated by the Matador SPL approach based on automatically identified features for three model variants from our *Body Comfort System (BCS)* running example in Section 2.2.

```
1  "AS":
2    <AS.decore>
3
4  "AutoPW":
5    <AutoPW.decore>
6
7  // ...
8
9  "FP_V2_V3_V7":
10   <FP_V2_V3_V7.decore>
```

Listing 5.4: Excerpt from a DeltaEcore mapping between features and corresponding delta modules generated by the Matador SPL approach for the DeltaEcore feature delta modules for the Central Locking System (CLS) variants from our running example in Section 2.2.

Delta Module to Feature Mapping Generation To allow derivation of variants from the generated SPL, the Matador SPL approach also generates a mapping between the features of the feature model and corresponding delta modules. This way users can easily create product configurations by selecting a feature from the feature model without knowledge of the underlying delta module. This is especially useful in cases where the generated SPL is extended and maintained in a distributed scenario (e.g., different departments developing different features). Thus, we generate an initial mapping file reducing the adoption effort for developers. In Listing 5.4, we present an excerpt from the DeltaEcore mapping file between the features of the feature model in Figure 5.2 and the corresponding delta modules. Such a mapping in turn also allows generation of product configurations for all analyzed variants in the 150% model.

Product Configuration Generation Based on the identified features with their information on the containing variants, the Matador SPL approach generates configurations for the input variants and, thus, allows direct derivation of the input variants from the newly generated SPL. In addition, developers can also create new configurations based on the generated SPL artifacts. In Listing 5.5, we show an exemplary DeltaEcore configuration for one of the variants that served as input for the feature model in Figure 5.2. Important to notice is that the configurations contains references to the `BCS_AutoPW` and `AutoPW` features that were not present in the original input variant as they are alternative to the `BCS_ManPW` and `ManPW` features, respectively. While this representation is counterintuitive, it is necessary due to fact that delta modules executing the removal of these features are needed. In a subsequent edit step, the users can rearrange the mappings to realize a more intuitive selection of such features (e.g., that selecting feature `ManPW` automatically removes the alternative feature `AutoPW`). To ease identification of such delta modules, the generation adds the prefix `REMOVE_` to the names of delta modules realizing only the removal of complete features.

Application Order Constraint Generation During the generation of delta modules, the Matador SPL approach uses the additional `requires` keyword for DeltaEcore delta modules to add dependencies between the different feature modules. In contrast to the generation of single delta modules per variant these additional dependencies are necessary to ensure the correct execution order of the delta modules. In general, features on the same hierarchy level of the generated feature model do not have any dependencies between each other. This is due to the fact that we use the model hier-

```
configuration <BCS.defeaturemodel> {
  "BCS",
  "BCS_AutoPW",
  "BCS_EM",
  "BCS_FP",
  "BCS_HMI",
  "BCS_ManPW",
  "AutoPW",
  "HMI",
  "EM",
  "FP",
  "FP_V1_V4_V5_V6",
  "ManPW"
}
```

Listing 5.5: DeltaEcore configuration generated by the Matador SPL approach for an exemplary variant based on the DeltaEcore feature model in Figure 5.2 for the Central Locking System (CLS) variants from our running example in Section 2.2.

archy of the 150% model to derive corresponding features. For dependencies across the hierarchy levels of the feature model, the approach uses the following rules for delta modules of features:

1. *bottom-up execution for removed features*: Features that are deeper in the model hierarchy have to be removed first. Thus, the execution starts with removed delta modules DM_r at the highest hierarchy level h and we add requires dependencies to their parent feature delta modules dm_p:

$$\forall dm_r \in DM_r \text{ holds } dm_r.dm_p \text{ requires } dm_r$$

2. *top-down execution for added features*: Features that are higher in the model hierarchy have to be added first. Thus, the execution starts with added features DM_a at the lowest hierarchy level h and we add requires dependencies to their child features DM_c:

$$\forall dm_a \in DM_a \text{ holds } \forall dm_c \in dm_a.DM_c \text{ requires } dm_a$$

Based on the added requires dependencies, DeltaEcore is capable of deriving a corresponding execution order. Furthermore, DeltaEcore allows users to explicitly define the execution order of their delta modules by using so-called application order constraints. Our Matador SPL approach also uses these facilities to make the dependencies between the delta modules explicit to the user in a single file. In Listing 5.6, we show the corresponding application order for the delta modules generated in accordance to the identified features in Figure 5.2. Each block enclosed in square brackets holds references to a number of DeltaEcore delta modules that can be executed in an arbitrary order inside this block. However, the blocks themselves have to be executed in a top-down order as specified by the application order file. As we can see, the feature identification mostly identified additive variability as mostly features with the `BCS_` prefix are added prior to their child features. This is due to the fact that the variant with the smallest number of features served as SPL core variant. However, also cases exist where functionality is removed, and we manually added corresponding comments to the generated application order constraints. In case of the `ManPW` feature, we can see

```
1  [
2    <BCS_AS.decore>,
3    <BCS_AutoPW.decore>,
4    <BCS_CLS.decore>,
5    <BCS_EM_Heat.decore>,
6    <BCS_LED.decore>,
7    <BCS_RCK.decore>,
8    <FP_V2_V3_V7.decore>,        // also removes "FP_V1_V4_V5_V6"
9    <HMI_Controller.decore>,     // also removes "HMI"
10   <REMOVE_ManPW.decore>        // removes "ManPW"
11 ]
12 [
13   <AS.decore>,
14   <AutoPW.decore>,
15   <REMOVE_BCS_ManPW.decore>,   // removes "BCS_ManPW"
16   <CLS.decore>,
17   <EM_Heat.decore>,
18   <LED.decore>,
19   <RCK.decore>
20 ]
21 [
22   <LED_AS.decore>,
23   <LED_EM_Heat.decore>,
24   <CLS_V1_V4_V6.decore>,
25   <CLS_V2.decore>,
26   <CLS_V3_V7.decore>
27 ]
28 [
29   <LED_AS_Active.decore>,
30   <LED_AS_Alarm.decore>,
31   <LED_AS_Alarm_Detected.decore>
32 ]
```

Listing 5.6: DeltaEcore application order constraints generated by the Matador SPL approach for the DeltaEcore feature delta modules for the Central Locking System (CLS) variants from our running example in Section 2.2.

that its leaf feature is removed prior to its parent feature. In case of the `FP` feature only a single delta module for variant `V2`, `V3` and `V7` exists as all other variants use the same implementation as the SPL core. Thus, the `FP_V2_V3_V7` delta module also removes the corresponding implementation if necessary. Similarly, the `HMI_Controller` removes also all unnecessary implementation details of the `HMI` feature.

5.4.3. Generating Recommendations for Feature Model Constraints

Constraints over features allow to restrict the number of legal variants that can be derived from an SPL or to define dependencies that cannot be expressed using the standard modeling capabilities of feature models (e.g., cross-tree constraints). Automatically deriving such constraints from a set of given variants is not an easy task as algorithms can only consider the given information and, thus, might not always derive constraints that are globally true. For instance, when analyzing a subset

of all variants, an algorithm might identify that certain features are never used together and, thus, conclude that they are mutually exclusive. However, considering all variants might reveal that they actually can coexist and, thus, the previously derived constraint is wrong. As a result, the Matador SPL approach analyzes the given variants to identify and generate constraints, *but* we only consider them as recommendations that have to be evaluated by domain experts. Furthermore, we do not claim that they represent a complete set of all possible constraints, but should only serve as initial guidance of experts towards potential relevant relations.

The Matador SPL approach is capable of identifying three types of potential constraints. First of all, it analyzes the combinations between optional features f_{o1} and f_{o2} by comparing the identifiers of their containing model variants ID_{mv} to identify the following two possible constraints:

- *Mutually Exclusive (XOR) Relations:* $f_{o1} \veebar f_{o2} \Leftrightarrow (\neg f_{o1} \vee \neg f_{o2}) \wedge (f_{o1} \vee f_{o2})$ holds iff we identify $\forall id_{mv} \in ID_{mv}$ that f_{o1} and f_{o2} are *never* used together, but $\forall id_{mv} \in ID_{mv}$ *either* f_{o1} *or* f_{o2} is included in each variant.
- *Optional Mutually Exclusive (XOR) Relations:* $\neg(f_{o1} \wedge f_{o2})$ holds iff we identify $\forall id_{mv} \in ID_{mv}$ that f_{o1}, f_{o2} or *none* of them is selected, but *never* both together.

As XOR relations are a more restrictive version of the optional XOR relations (i.e., they require that one the of the features is selected in each model variant), we only generate optional XOR constraints in cases the Matador SPL approach did not identify a corresponding XOR relation. This reduces the number of generated constraints and, thus, eases analysis by experts.

In addition to the previous constraints, we analyze the relations between sets of optional child features F_{oc} and their common parent feature f_p (independent of its variability):

- *Requires Relations:* $f_{oc} \Rightarrow f_p$ holds iff we identify $\forall id_{mv} \in ID_{mv}$ that $f_{oc} \in F_{oc}$ requires f_p As multiple children might exist below f_p, we combine the corresponding requires relations into $(f_{oc_1} \vee f_{oc_2} \vee \ldots \vee f_{oc_n}) \Rightarrow f_p$ to reduce the number of generated constraints.

Obviously, these constraints will not be identified if the user activates the option to reduce the variability of optional child features to mandatory in cases where they are contained in the same model variants as their parent feature (cf. Section 5.4.2). Furthermore, it would, in theory, be possible to generate *OR relations* (e.g., $f_1 \vee f_2$). However, as the identified features in the feature model stem from the input variants of the Family Mining algorithm, they would not exist in the generated feature model if they were not contained in at least one of the input variants. Thus, such constraints apply to almost all combinations of features as only requirement for an OR relation is that one of the features is selected in one of the analyzed variants. As a result, we decided against including these constraints in our recommendations as they would overwhelm developers with a flood of additional and potentially meaningless constraints. For instance, from a logical point of view, it might be true that the analyzed variants represented by the generated feature diagram in Figure 5.2 have an OR relation between the feature BCS_AS and BCS_EM. However, this relation exists by accident as the mandatory feature BCS_EM represents the exterior mirror, while the feature BCS_AS is not related with this functionality and represents the optional alarm system.

In Listing 5.7, we present an excerpt from the DeltaEcore constraints generated for the feature model in Figure 5.2. These constraints require manual analysis to remove constraints that unnecessarily restrict the variability. As we can see, the algorithm correctly identified the requires relations

```
1  "LED" -> "BCS_LED"
2  ("LED_AS" || "LED_EM_Heat") -> "LED"
3  ("LED_AS_Alarm" || "LED_AS_Alarm_Detected" ||
4      "LED_AS_Alarm_Active") -> "LED_AS"
5  "ManPW" -> "BCS_ManPW"
6  "AutoPW" -> "BCS_AutoPW"
7  "EM_Heat" -> "BCS_EM_Heat"
8  "AS" -> "BCS_AS"
9  "CLS" -> "BCS_CLS"
10 "RCK" -> "BCS_RCK"
11
12 (!"BCS_ManPW" || !"BCS_AutoPW") && ("BCS_ManPW" || "BCS_AutoPW")
13 (!"ManPW" || !"AutoPW") && ("ManPW" || "AutoPW")
14 (!"HMI" || !"HMI_Controller") && ("HMI" || "HMI_Controller")
```

Listing 5.7: Excerpt from the DeltaEcore cross-tree constraint recommendations generated by the Matador SPL approach for the DeltaEcore feature model in Figure 5.2 for the Central Locking System (CLS) variants from our running example in Section 2.2.

between the features using the `BCS_` prefix and their child features. In this concrete example, most of these constraints are an indicator for potential to merge the children in their corresponding parent feature. During the merging of the 150% model that served as input to the feature identification the `ManPW` and `AutoPW` regions where not merged as alternatives, because their names are not equal enough according to the Levenshtein Distance algorithm. Thus, the merge algorithm split the corresponding comparison element into two optional parts. As a result, the variability between these features is not restricted enough and, in theory, variants with both features are feasible. This missing information is now pointed out to the user through the constraint recommendations generated by the Matador SPL algorithm. Thus, our approach uses a optimistic solution and rather allows for more legal variants than the analyzed input variants (i.e., combinations of the two features) instead of restricting the generated SPL too much. However, the Matador SPL approach tries to point out such situations and, thus, eases their identification by domain experts. In this concrete example, users might already have resolved this "issue" by changing the variability for two parts during an analysis of the 150% model prior to the execution of the Matador SPL approach. In this case, the algorithm would already generate a corresponding alternative group. Or they might conclude from the generated constraint that such an alternative between the features is sensible and change this in a subsequent refactoring step of the generated SPL by removing the constraint and adding a direct alternative in the feature model.

5.5. Restructuring the Automatically Generated Software Product Line

To support developers during the refactoring of the generated SPL, our Matador SPL approach provides additional operations for applying consistent changes to the SPL. While other refactoring approaches exist, such as [AGM+06] or [SRS13], we base our solutions on work by Seidl et al. [SHA12], who propose generic operators to realize consistent co-evolution of SPL artifacts and feature mappings. Main reason for selecting the approach by Seidl et al. [SHA12] is their clear discussion of the

implications for the different problem and solution space artifacts when executing the proposed refactorings. Furthermore, Seidl et al. [SHA12] show appropriate solutions to involve users in the refactorings. For example, when moving mappings of features, the authors allow to involve user knowledge to select subsets of mappings that should be assigned to features. For a detailed discussion of these operators proposed by Seidl et al. [SHA12], we refer to [SHA12] and only give a brief description of each operator. The authors propose eight operators to modify existing mappings between features and corresponding implementation artifacts:

- *Feature Remapping Operators:* These operators allow consistent refactoring of feature mappings (i.e., mappings of delta modules M_{dm} to features in our case) when modifying features in our generated feature model:
  - *Move Feature Mapping:* This operator moves the mappings M_{dm} of a feature f_A to another feature f_B. After the execution the mappings for f_A are empty and f_B holds the corresponding mappings M_{dm}.
  - *Copy Feature Mapping:* This operator copies the mappings M_{dm} of a feature f_A to another feature f_B. After the execution the mappings for f_A and f_B both hold the mappings M_{dm}.
  - *Remove Feature Mapping:* This operator removes the mappings M_{dm} from a feature f. After the execution the mappings for f are empty.
  - *Split Feature Mapping:* This operator splits the mappings M_{dm} of a feature f to multiple other features $f_1, f_2, \ldots, f_n$. After the execution the mappings for f are empty and $f_1, f_2, \ldots, f_n$ each hold a subset $M_{dm_s} \subset M_{dm}$ of these mappings specified by a user selection. In case such a selection is not available, $f_1, f_2, \ldots, f_n$ each hold the complete mappings M_{dm}.
  - *Merge Feature Mapping:* This operator merges the mappings $\{M_{dm_1}, M_{dm_2}, \ldots, M_{dm_n}\}$ from features $f_1, f_2, \ldots, f_n$ to feature f. After the execution the mappings for $f_1, f_2, \ldots, f_n$ are empty and f holds a superset $M_{dm_s} \supset \{M_{dm_1}, M_{dm_2}, \ldots, M_{dm_n}\}$ of these mappings.
- *Object Remapping Operators:* These operators allow consistent refactoring of object mappings (i.e., delta modules storing the SPL implementation in our case) when modifying the implementation of the SPL:
  - *Move Object Mapping:* This operator moves the feature mappings M_f of a delta module dm_A to another delta module dm_B. After the execution the mappings for dm_A are empty and dm_B holds the corresponding mappings M_f.
  - *Copy Object Mapping:* This operator copies the feature mappings M_f of a delta module dm_A to another delta module dm_B. After the execution the mappings for dm_A and dm_B both hold the mappings M_f.
  - *Remove Object Mapping:* This operator removes the feature mappings M_f from a delta module dm_A. After the execution the mappings for dm_A are empty.

Based on these operators, we realized the following operators, which not only preserve consistency between features and their corresponding delta modules as in [SHA12], but also modify configurations, constraints and the application order of delta modules where applicable. All operators

are summarized in Table 5.3 together with a reference showing their application on a corresponding example in Figure 5.3. In cases where we cannot provide an automatic operation to preserve the consistency (e.g., due to missing information) or where the operation might change the behavior of the SPL (e.g., when removing a feature from the feature model and corresponding configurations), we generate corresponding warnings to make the user aware of such situations.

- *Add Feature:* Adds a feature f to the feature model. Users can select delta modules mapping to f and configurations that should include f. Warnings are generated as the new feature might require modifications to the application order or the requires relations of the selected delta modules, which cannot automatically be derived.
- *Remove Feature:* Removes a feature f from the feature model, mappings, constraints and configurations. Warnings are generated as the delta modules mapped to the removed feature might still be indirectly applied due to requires relations of other delta modules.
- *Rename Feature:* Renames all occurrences of a feature f in the SPL.
- *Move Feature:* Moves a feature f together with its sub tree of features from its current parent feature f_p^c to a new parent feature f_p^n. Warnings are generated for configurations, constraints, application order and requires relations as moving a feature in the feature model hierarchy can have severe impact on its dependencies. For instance, constraints can become redundant if they are already expressed by the feature model.
- *Change Variability:* Allows to change the variability for a feature f to mandatory, optional or alternative. In case the feature is changed to mandatory, it is added to all configurations if necessary (i.e., if the path to the feature model root contains only mandatory features) that do not already contain the feature. Warnings are generated as constraints can become invalid during the change of variability. For instance, changing a feature to mandatory can make a mutual exclusion constraint invalid.
- *Create Alternative / Or:* Allows to create an alternative / or group from a set of selected features $\{f_1, f_2, \ldots, f_n\}$. Warnings are generated for configurations, constraints, application order and requires relations as creating such groups can have severe impact on feature dependencies. For instance, existing configurations containing multiple of the alternative features become invalid due to the introduced constraint.
- *Refine Feature:* Allows to refine the implementation of an existing feature f by adding corresponding child features $\{f_1, f_2, \ldots, f_n\}$. Users can select delta modules mapping to features $\{f_1, f_2, \ldots, f_n\}$ and configurations that should include them. Warnings are generated as the new features might require modifications to the application order or the requires relations of the selected delta modules, which cannot automatically be derived.
- *Merge Into:* Allows to merge sub features $\{f_1, f_2, \ldots, f_n\}$ into their parent feature f_p. Each feature $\{f_1, f_2, \ldots, f_n\}$ is removed from the feature model and all configurations. We distinguish between *shallow* and *deep* merging. In case of *shallow* merging each feature $f \in \{f_1, f_2, \ldots, f_n\}$ is merged into its parent feature f_p and the corresponding child features of f get f_p assigned as their new parent. In case of *deep* merging the complete subtrees below each feature $f \in \{f_1, f_2, \ldots, f_n\}$ are merged into feature f_p. All mappings for the features are merged

into the mapping of parent feature f_p and constraints comprising one of the features are removed. Warnings are generated as the delta modules mappings merged into the mapping of parent feature f_p might require changes to the requires relations of other delta modules or the application order. Furthermore, warnings are generated to check the configurations as configurations that previously only contained feature f_p now might unintentionally also contain the functionality of the merged features.

- *Copy:* Creates a copy f_c of feature f with all its constraints and mappings. In case of copying a mandatory feature, it is added to all existing configurations respecting the feature model (i.e., if the path to the feature model root contains only mandatory features). Otherwise, users can select configurations that should include f_c. We distinguish between *shallow* and *deep* copying. In case of *shallow* copying, only feature f is copied. In case of *deep* copying, feature f is copied together with its complete subtree.

Using the described operators, we are able to refactor the SPL generated by our Matador SPL approach in Figure 5.2 from a realization that is close to the original implementation into a form that meets more the expectations of an SPL engineer. In Figure 5.4, we present the resulting SPL. The following list shows an excerpt from the operators that we used to create the corresponding result and discusses exemplary applications for them. For a complete list of all executed refactoring operations, we refer to Appendix E.

- *Merge Into (Shallow):* By applying this operator, we remove unnecessary hierarchical decomposition. For instance, we are able to merge features with their parent feature containing the `BCS_` prefix (e.g., feature `ManPW` into `BCS_ManPW`). During its execution, the refactoring operator ensures that the mapping of the merged feature is correctly assigned to its corresponding parent feature.

- *Rename:* Afterwards, we are able to rename the feature and assign a more descriptive name to it or at least remove the `BCS_` prefix originating from the implementation analyzed during mining. The operator ensures that all SPL artifacts referencing the renamed feature are modified accordingly.

- *Remove Feature with Artifacts:* This operator is applied to remove the `HMI_Controller` feature together with its corresponding delta module as it realizes the same functionality as the `HMI` feature (cf. Section 5.4.2). During the execution, the operator removes all references to the feature and its delta modules from the SPL artifacts.

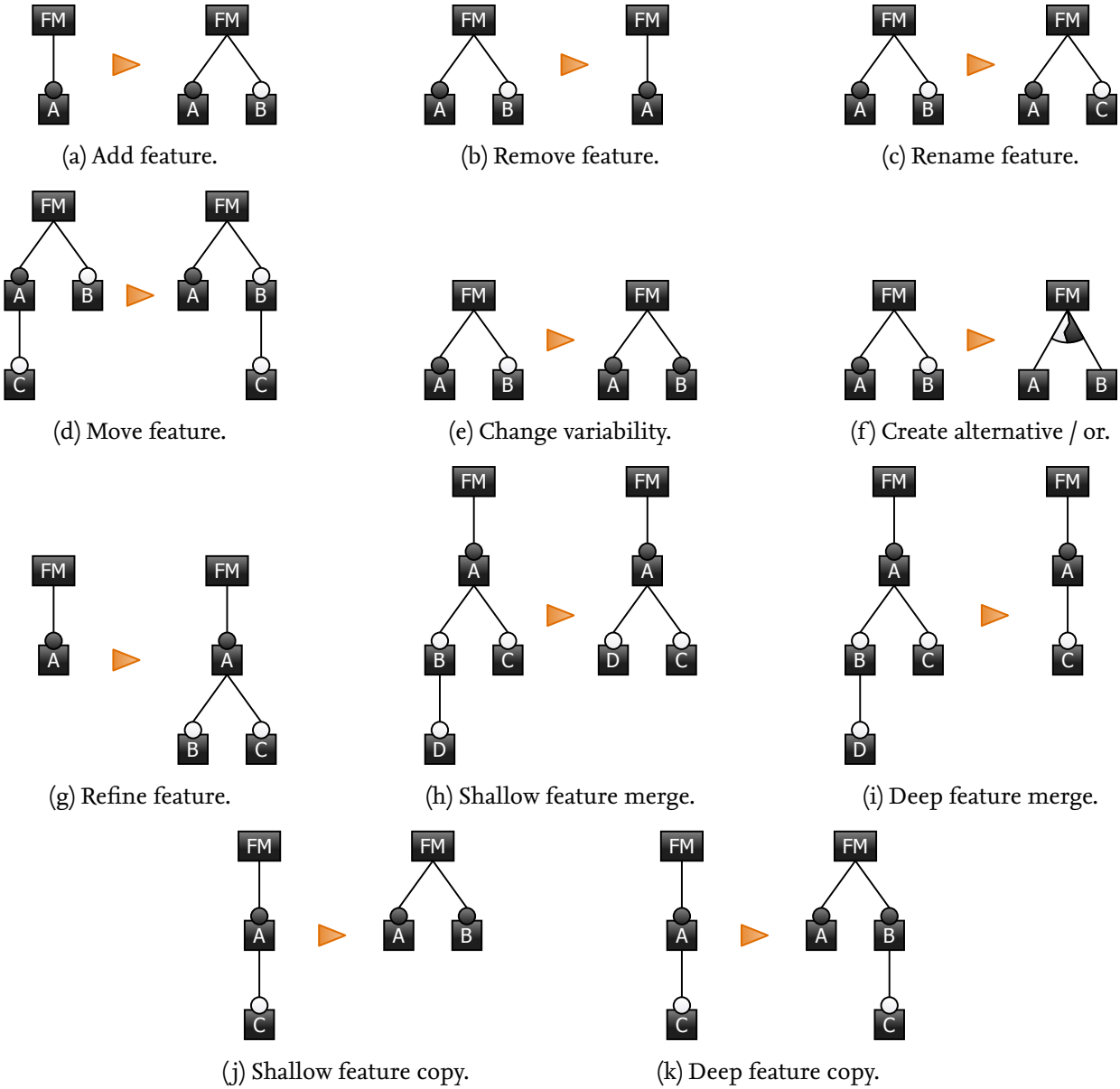
- *Add Feature:* This operator is used to introduce the `PW`, `Security` and `Door System` features to the SPL, which are later used to group other features below them.

- *Change Variability:* This operator is used to change the variability of the introduced `PW` and `Door System` features from optional to mandatory. This change of variability is necessary to include the features in all variants as new features are added as optional by default in order to not be too restrictive (i.e., force their inclusion in all variants). During the execution, the operator automatically updates the configuration for the variants and adds the new mandatory feature to their selection if necessary.

	Add Feature	Remove Feature	Rename Feature	Move Feature	Change Variability	Create Alternative/Or	Refine Feature	Merge Into Feature		Copy Feature	
								Shallow	*Deep*	*Shallow*	*Deep*
Feature Model	added	removed***	renamed	moved	change variability	create alternative	add sub features	merged	merged incl. sub features	copied	copied with sub features
Mapping	select	removed	renamed	–	–	–	select	merged	merged	copied	copied
Constraints	–	removed	renamed	warning	warning	warning	–	removed	removed	copied*	copied*
Application Order	warning	warning	–	warning	–	warning	warning	warning	warning	–	–
Configurations	select	removed*	renamed	warning	added**	warning	select	removed*	removed*	added** / select	added** / select
Delta Module	warning	warning	–	warning	–	warning	warning	warning	warning	–	–
Example Figure	5.3a	5.3b	5.3c	5.3d	5.3e	5.3f	5.3g	5.3h	5.3i	5.3j	5.3k

* = *additional warnings generated to manually check refactoring* ** = *for mandatory features* *** = *also option to remove corresponding delta modules available*

Table 5.3.: Steps executed for the different SPL artifacts during the refactoring operations provided by the Matador SPL approach.



(a) Add feature. (b) Remove feature. (c) Rename feature.

(d) Move feature. (e) Change variability. (f) Create alternative / or.

(g) Refine feature. (h) Shallow feature merge. (i) Deep feature merge.

(j) Shallow feature copy. (k) Deep feature copy.

Figure 5.3.: Operations provided by the Matador SPL approach to manually refactor the generated SPL.

- *Move Feature:* This operator is used to move the `EM_Heat` feature (renamed to `Heatable` in the final SPL) below the `EM` feature as it realizes extended functionality for the exterior mirror. During the execution, the operator generates warnings to make users aware of the impact on other SPL artifacts that cannot be automatically updated. For example, the impact of this move operation on existing constraints cannot be estimated and we, thus, leave corresponding checks to a manual inspection by users.
- *Create Alternative:* As the `ManPW` feature and the `AutoPW` are alternative to each other, we introduced an alternative group between them as suggested by the, afterwards removed, constraint in Listing 5.7. This makes the variability relation between these features visually explicit to the users of the SPL.
- *Create Or:* Using available domain knowledge, the user identified and added that the `LED AS` feature and the `LED Heatable` feature form an or group as at least one of them has to be included in variants if their parent feature `LED` is selected.

In addition to these operations, the constraints for the SPL are modified to reflect that the `LED AS` and `LED Heatable` features have to be selected with the corresponding features controlling their main functionality (i.e., the `AS` and `Heatable` features). In addition, modifications to the generated delta modules were applied. For instance, the extracted delta module for the `CLS_V2` feature corresponds to an implementation that uses the `CLS` together with the `RCK` and `AutoPW` features. As for the analyzed variants the `RCK` feature is always used together with the `AutoPW` feature, it was possible to reduce the corresponding delta module to operations that are specific to the `RCK` feature in combination with the `AutoPW` feature. The remaining functionality will be added in any case by the `CLS AutoPW` implementation of the the `CLS` feature. To ensure this behavior, we added the additional cross-tree constraint `RCK -> CLS AutoPW`.

When comparing the refactored feature model (cf. Figure 5.4) with the original *BCS* feature model (cf. Figure 2.9), we can see that less differences exist compared to the previously generated feature model (cf. Figure 5.2). Apart from the fact that only a subset of all *BCS* features is included in the feature model in Figure 5.4 and we introduced the additional `RCK -> CLS AutoPW` constraint, the only noticeable difference is that the additional alternative `CLS ManPW` and `CLS AutoPW` features exist with corresponding cross-tree constraints. The reason is that in case of the original *BCS* case study a delta module with the shared functionality of the `CLS ManPW` and `CLS AutoPW` features exists and their differing functionality is added by a separate application condition checking for the presence of the `ManPW` or `AutoPW` feature. Thus, the original *BCS* implementation uses an implicit solution (i.e., the application condition) to add the differences, while our automatically identified and refactored feature model uses an explicit solution (i.e., the alternative features visible to the user). Overall, although being modeled differently, the identified variability of the created SPL allows behavior that conforms to the ground truth of the original *BCS* case study.

One can argue that manual refactorings of the generated SPL are undesirable as they require additional effort by the user. However, the Matador SPL approach already executes most of the migration for the input implementations to an SPL automatically (i.e., all relevant artifacts are generated). Furthermore, the refactoring of such generated SPLs is largely automated by our operators and, thus, our Matador SPL approach supports the user during this step. In fact, we argue that automatically deriving the "perfect" SPL from an existing implementation is impossible as too many

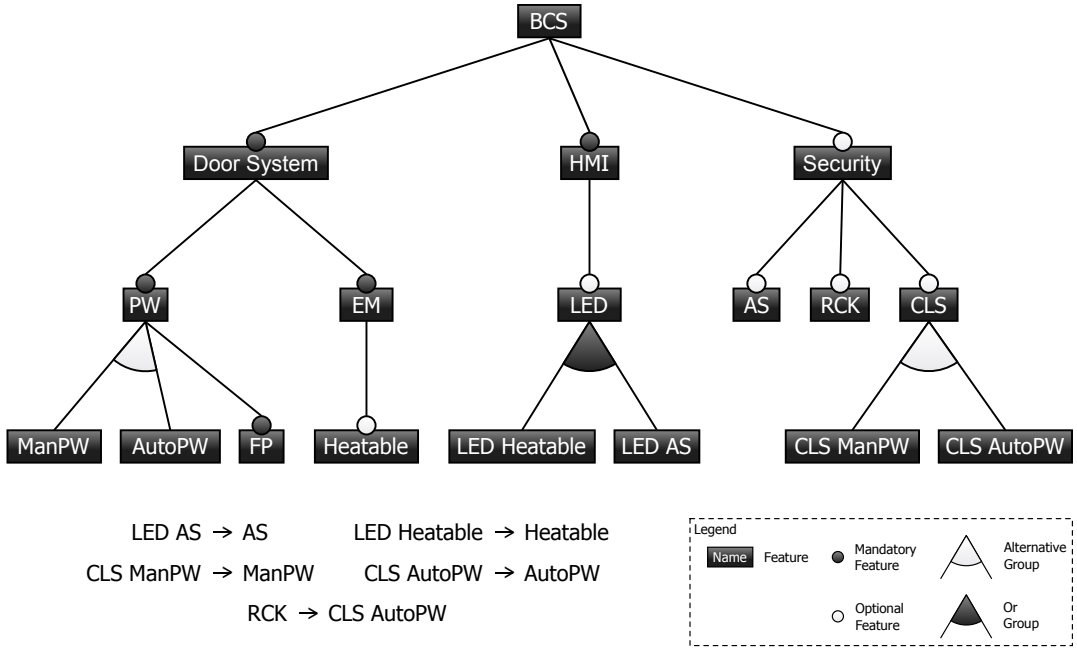


Figure 5.4.: Exemplary refactoring result based on the operators from Table 5.3 applied to the feature model generated by the Matador SPL approach in Figure 5.2.

user-specific factors have to be considered (e.g., the structuring of the SPL). Thus, we see the semi-automatic restructuring of the generated SPL with sound refactoring operators as an indispensable step during migration to an SPL conforming to the requirements of a company. Overall, our operators provide necessary tool support and easily allow to refactor the generated SPL.

5.6. Related Work

The work related with the Matador SPL approach consists of four major research areas:

- *Transformation Languages:* Our Matador SPL approach is able to automatically generate Delta-Ecore delta dialects. These can be used to derive corresponding delta languages and encode variability for the used modeling language. Other approaches exist to automatically derive languages with and without the presence of variability and we discuss them in Section 5.6.1.
- *Feature Location Techniques:* Our Matador SPL approach exploits the hierarchical decomposition of analyzed model variants to identify features that are closely related to the original implementation. Other approaches exist to identify features in source code as well as models and we discuss them in Section 5.6.2.
- *Automatically Encoding Variability in Software Product Lines:* The goal of our Matador SPL approach is to automatically encode identified variability from existing variants in a corresponding SPL realization. Other approaches exist to realize such an automatic transition to an SPL and we discuss them in Section 5.6.3.
- *Restructuring Software Product Lines:* The automatically generated SPL realization can be transformed using the operators provided by our Matador SPL approach. Other approaches exist to refactor SPLs and we discuss them in Section 5.6.4.

5.6.1. Transformation Languages

To allow creation of SPLs for different model-based languages, we apply delta-oriented transformation languages that we generate based on the identified variability of analyzed variants. Thus, the derived transformation languages are specifically tailored towards the applied language and, furthermore, support handling of variability in SPLs. In literature, a variety of languages exist to transform existing implementation artifacts or even to define and generate domain-specific transformation languages. These range from general purpose transformation languages to languages specifically providing transformation facilities in the presence of variability.

Source Code Transformation Languages Source code transformation languages have been research target for a long time [PS83] and also different modern approaches exist. The TXL allows definition of grammars to parse source code of different languages and to transform it using rule based transformations [Coro6]. Spoon[3] is based on a meta-model representation that is used to represent parsed Java source code [PMP+15]. Users can write their own filter and transformation rules for the parsed source code by using methods from the Java API of Spoon that were specifically created for this purpose. Stratego[4] integrated in the Spoofax language workbench[5] provides source code transformations based on rewriting strategies, which allow to separate application strategies from the actual transformation rules [Viso1]. As a result, transformation rules can easily be reused across different transformations. Rascal[6] combines transformation techniques similar to these approaches with additional capabilities for analyzing source code [KSV09]. This way, the authors allow developers to integrate source code analysis and transformation in their projects without using separate solutions (e.g., based on two separate libraries) with different language designs. Thus, the effort to integrate and apply these techniques is reduced as only a single language has to be understood and used. In addition, Hölldobler et al. [HRW15] describe a tool-supported approach to systematically derive DSL-specific transformation languages that do not rely on generic syntax, but use keywords close to the concrete syntax of the original DSL. This way, developers can more easily understand and express needed transformations in the corresponding language. Overall, these general purpose and domain-specific transformation languages are theoretically capable of handling source code transformations in the presence of variability and, thus, could be used in an SPL. However, as they were not specifically designed as variability realization mechanisms, additional logic would be necessary to correctly realize transformations following SPL artifacts, such as feature models with corresponding configurations.

In contrast, other transformation languages exist that specifically provide facilities to realize SPLs using source code. Examples are aspect-oriented languages, such as AspectJ[7] [KHH+01] and AspectC++[8] [SGS02], or delta-oriented languages, such as DeltaJ [KHS+14, SBD11]. Following the principles of AOP, these aspect-oriented source code languages allow to define pointcuts (e.g., for specific methods) with corresponding advices (e.g., extensions of the method body) that are executed when selected. Similarly, DeltaJ extends plain Java with additional constructs to support DOP by

[3] http://spoon.gforge.inria.fr/
[4] http://www.strategoxt.org/
[5] http://www.spoofax.org/
[6] https://www.rascal-mpl.org/
[7] https://www.eclipse.org/aspectj/
[8] https://www.aspectc.org/

allowing definition of delta modules to add (e.g., complete classes), remove (e.g., methods) or modify (e.g., attributes of classes) source code from existing programs. While these languages explicitly support encoding of variability for an SPL, they are specifically tailored towards the underlying language. As a result, they are not suitable for our purposes as we provide variability mining for different block-based languages that might not have an existing variability language.

To overcome this limitation, Haber et al. [HHK+13, HHK+15] describe an approach to generate delta languages from grammars of textual languages together with a common delta language provided by the authors' tooling. While the approach allows to easily derive new delta languages, it does not provide generation facilities for corresponding tooling, such as tools allowing actual product derivation from the defined deltas, type checking or editor support. In contrast, our approach relies on DeltaEcore by Seidl et al. [SSA14] (cf. Section 2.3.6) to easily define and derive fully functional variability languages that allow derivation of variants through DeltaEcore's facilities. Thus, we can directly utilize these languages during encoding of the identified variability in an SPL and developers can utilize the facilities provided by DeltaEcore (e.g., type checking or editor support).

Model Transformation Languages Similar to source code transformation languages, a variety of general purpose model transformation languages exists [CH06]. The *Atlas Transformation Language (ATL)*[9] uses side-effect free helper methods and rules to match and transform model elements [JK06]. The *Query/View/Transformation (QVT)* languages conforming to the standard specified by the OMG are executed on models conforming to the MOF standard [Obj16a] and not only allow transformations but provide means to select model elements and show views on them [Obj16b]. Furthermore, the standard allows for unidirectional and bidirectional transformations between models. The *Epsilon Transformation Language (ETL)*[10] was developed as many existing transformation languages are specifically implemented to execute transformations and do not provide integration with other tooling (e.g., model validation or model merging) [KPP08]. By building the ETL on top of their Epsilon Model Management Platform, the authors are capable to provide users with a large variety of tools using similar syntax and common infrastructures [KPR+11]. Thus, the overall effort of using these tools in an integrated manner is reduced. In contrast to these languages, Schmidt [Sch07] describes an approach that allows to define model transformations for UML models [Obj15] using patterns written in the original UML syntax. Additionally, Rumpe et al. [RW11] describe an approach to reuse concrete syntax elements from DSLs in corresponding transformation languages and even generate such languages automatically. Although allowing generation of domain-specific transformation languages, this approach does not support generation of languages that support management of variability. Thus, similar to the other general purpose model transformation languages it would require additional logic to cope with transformations in the presence of variability.

In contrast, similar to textual transformation languages, also other model transformation languages exist that were specifically designed to consider variability. Examples are the CVL [HMO+08], and delta-oriented languages, such as DeltaSimulink [HKM+13] for Matlab/Simulink, Delta Rhapsody for IBM Rhapsody [BSE+16], Δ-MontiArc [HKR+11] for the architecture description language MontiArc[11] [HRR12] and the UML-DOP profile [SHM+16] for the UML [Obj15]. The CVL is a generic language that allows to derive variants for any MOF-based model by defining corre-

[9] http://www.eclipse.org/atl/
[10] http://www.eclipse.org/epsilon/
[11] http://www.monticore.de/languages/montiarc/

sponding variation points that are resolved with selected replacements. While the CVL allows explicit modeling of variability for different model-based languages, we decided to use delta modeling for the SPL realizations generated by the Matador SPL approach due to its high flexibility and modularity (cf. Section 2.3.6). The mentioned delta-oriented languages adopt the ideas of DOP for the corresponding languages and, thus, allow to add, remove or modify their language artifacts in an SPL. While these languages allow explicit modeling of variability for their corresponding "parent" language, they are limited exactly to it and cannot be applied to other languages or dialects.

To overcome this limitation, Zschaler et al. [SLF+08, ZSS+09] describe an approach to allow derivation of variability languages that rely on a general purpose transformation language (i.e., xTend[12]). This approach does not provide a common variability language and developers have to reimplement the different operations for each new language. In contrast, DeltaEcore by Seidl et al. [SSA14] reduces the effort for developers during creation of new delta languages as it is capable of generating complete languages based on a user-provided meta-model and its common variability language. As our Matador SPL approach relies on these facilities, it exploits these techniques and similarly does not need the user to provide additional manual input to derive a delta language.

5.6.2. Feature Location Techniques

In literature, a variety of algorithms exists to reverse-engineer features and feature models from different artifacts. While approaches exist to derive feature models from requirements or product descriptions, also approaches exist to locate features in source code or modeling artifacts.

Feature Location Techniques for Requirements Different algorithms exist to locate features in requirements of software systems. Examples are the approaches by Li et al. [LSS18], Chen et al. [CZZ+05], Loughran et al. [LSR05] and Weston et al. [WCR09]. These approaches use different NLP techniques to process input requirements documents and to identify potential features. For instance, the approach by Li et al. [LSS18] uses unsupervised learning (i.e., no user input is required and the system learns based on the input data) to automatically derive features. First, the pre-processed input documents (e.g., stopwords are removed) are translated to matrices that are used to represent the analyzed requirements with vectors. Based on this data the authors train a neural network, which later allows to apply clustering on the learned characteristics of the requirements (e.g., k-means) to identify sentences that belong to the same cluster (i.e., feature). While the different approaches are capable of identifying potential features in natural language requirements, they do not consider implementation artifacts, such as source code or models. In contrast, we are interested in concrete mappings of implementation artifacts to features. As a result, the discussed approaches are not directly applicable for our Matador SPL approach.

Feature Location Techniques for Product Descriptions A number of different techniques exist to locate features and generate feature models from more or less formal product descriptions. Approaches exist that consider informal product descriptions in form of comparison tables (i.e., listing the characteristics of the products) as input and allow automatic generation of corresponding feature models. Examples are the approaches by Acher et al. [ACP+12], Davril et al. [DDH+13] and Bécan et al. [BBG+15]. For instance, Acher et al. [ACP+12] use their language VariCell to allow developers to guide the parsing (e.g., to scope the imported data) and processing (e.g., to define the

[12] http://www.eclipse.org/xtend/

hierarchy levels of features or which values form alternatives) of *Comma-Separated Values (CSV)*-files prior to the generation of a corresponding feature model. Based on the imported data the authors generate one feature model per contained product and merge these afterwards into a single feature model for all products. In addition, Ferrari et al. [FSD13] describe an approach to suggest features for related products from public documents describing their functionality (e.g., product brochures) using NLP techniques. This approach is particularly interesting when executing a domain analysis for a product and comparing existing products from competitors.

Further techniques exist to automatically derive feature models from logical formulas. These techniques rely on the fact that feature models and constraints between features can be expressed in propositional logic [Bat05]. Examples are the approaches by She et al. [ACS+12, SRA+14] and Czarnecki et al. [CW07]. For instance, Czarnecki et al. [CW07] build an implication graph (i.e., a graph showing the dependencies between features) from an input formula. Based on this graph the authors execute a transitive reduction to identify clear parent-child relations between the features. The features' variability is derived by further analyzes of the implication graph. For instance, mandatory features can be identified by searching for cliques (i.e., graph nodes with edges to all other nodes of the clique) in the graph. In addition, She et al. [SLB+11] describe a similar approach that uses not only propositional formulas, but also descriptions of the contained features. This way, the authors are capable of providing developers with a ranked list of potential parent-child relations for features by analyzing the different feature descriptions using NLP techniques.

Other techniques generate feature models from existing product configurations (e.g., feature selections or incidence matrices) without additional input on the features' relations (e.g., propositional formulas). Examples are the approaches by Haslinger et al. [HLE11, HLE13], Ryssel et al. [RPK11, Rys14], Czarnecki et al. [CSW08], Assunção et al. [ALL+15, ALL+17a] and Lopez-Herrejon et al. [LGB+12, LLG+15]. For instance, Haslinger et al. [HLE13] use a combination of feature implication graphs and mutual exclusion graphs (i.e., graphs showing features that are never selected together) to derive feature models with cross tree constraints. Another approach by Lopez-Herrejon et al. [LGB+12, LLG+15] uses different evolutionary algorithms to synthesize feature models from configuration tables. A special case represents the approach by Czarnecki et al. [CSW08] as it derives probabilistic feature models. These specialized feature models not only describe the problem space of variants in an SPL, but also store the probability of features to occur in any user-selected configuration. The authors use association rule mining to reverse-engineer such feature models from input configurations and calculate corresponding probabilities. Another special case is the approach by Ryssel et al. [RPK12b], which allows generation of feature models for *derived features*. These features are not directly user-selectable and are only implicitly activated (e.g., based on evaluated constraints in the feature model) [BSL+10]. The approach uses ontologies to express features with their selection conditions (e.g., that specific parameters are set to a certain value) and derives corresponding feature models. Furthermore, it is capable of verifying the correct selection of corresponding derived features in such systems.

Bécan et al. [BAB+16] describe an approach that combines implication graphs with ontological details of the feature names (e.g., by identifying relations between them using WordNet) to identify appropriate hierarchy levels for the generated feature model. Their approach is integrated in the interactive web tool WebFML by Bécan et al. [BBA+14], which allows to derive feature models from various artifacts (e.g., propositional formulas, product comparison matrices or dependency graphs).

Although the described approaches are capable of deriving features and feature models from different input artifacts, they do not consider concrete implementation artifacts from the solution space and mostly rely on heuristics to identify the hierarchical decomposition of features. In contrast, we are interested in linking the identified features with corresponding implementations. Thus, these algorithms are not suited for our goals without further adaptations. Furthermore, our approach does not have to rely on heuristics to identify the feature hierarchies but exploits the hierarchical structure of the analyzed modeling artifacts.

Feature Location Techniques for Software Histories In addition to the previously mentioned approaches, Li et al. [LZR+17] describe an approach that allows to locate features in the version history of software. The approach relies on test suites for the searched features as starting point for the location and identifies commits relevant to the features using slicing on the version history. Based on the identified information, the approach generates feature models expressing the relations between features during runtime. This information can be used by developers to understand the relations between features during the evolution of the software. As this approach follows a different goal than our Matador SPL approach (i.e., showing the dependencies between features in the presence of evolution), it is not applicable for the migrating of products to an SPL.

Feature Location Techniques for Source Code The larger part of existing feature location techniques concentrates on source code and a variety of corresponding approaches exists [AV14, DRG+11, RC13b]. For our discussion of existing source code feature location techniques, we follow the classification into *static* and *dynamic* approaches used by Rubin et al. [RC13b]. While static approaches concentrate on analyzing existing artifacts using, for example, NLP techniques, dynamic approaches use information on the executed code of the features (e.g., execution traces).

Examples for static feature location are approaches by Chen et al. [CCW+01], Peng et al. [PXT+13], Walkinshaw et al. [WRW07], Duszynski et al. [DKB11], Ballarín et al. [BLC16], Kästner et al. [KDO11, KDO14], Damaševičius et al. [DPK+12] and Ziadi et al. [ZFS+12]. For instance, Ballarín et al. [BLC16] apply feature location techniques to models that served as basis for either manually implemented or automatically generated source code. Their approach first uses an algorithm to identify features in the models based on user-provided seed fragments that serve as input to a genetic algorithm. Based on the identified model features, the authors use differencing algorithms on the corresponding source code and identify isolations of the features source code together with similarity values showing the relation between the same features in different products. Another approach by Walkinshaw et al. [WRW07] constructs so-called *Hammocks graphs* from user-specified *landmark* and *barrier* elements in the analyzed source code to identify features. While landmarks represent elements that are key to the corresponding feature (e.g., a key method), the barrier elements represent irrelevant parts for the analyzed feature. The hammock graph is built from the initial landmarks by using slicing to link any other elements between the landmarks using directed edges and pruning the resulting graph based on the barrier elements. This way, the authors are able to identify all elements belonging to a certain feature starting from user-specified seed elements. In addition, Lapeña et al. [LFP+16] use a different approach to locating features in source code. The authors assume that requirements exist for the analyzed product variants and allow users to formulate their queries in natural language. Using their Natural Language Query to Requirements Query (NLQ2RQ) approach, the authors search for requirements that have a strong relation to the natural

language query and automatically generate corresponding queries to search the source code for the feature based on these requirements. Thus, user input closer to the requirements is automatically translated to corresponding source code queries and only a search space limited to linked source code has to be analyzed.

Examples for dynamic feature location techniques are approaches by Olszak et al. [OJ09, OJ12], Savage et al. [SRP10], Eaddy et al. [EAA+08] and Anwikar et al. [ANC+12]. Olszak et al. [OJ09, OJ12] allow developers to define so-called *feature entry points* into object-oriented source code and use them to record feature traces when executing the corresponding source code. Based on these feature traces, the authors allow automatic restructuring of classes back and forth between a feature-centric view (i.e., a package structure explicitly highlighting features) and the original package structure. Another approach by Eaddy et al. [EAA+08] combines static and dynamic feature location techniques. The authors' Cerberus approach combines three techniques to analyze systems from different angles and derive feature locations with more certainty. The authors 1) apply IR techniques to the source code of analyzed programs and calculate the relevance of the identified terms, 2) use execution traces to identify methods and fields used by a specific feature and 3) use pruning to identify source code artifacts related to provided seed fragments for features. By aggregating the results from these techniques, the Cerberus approach is able to derive relevant elements for searched features.

Furthermore, work by Al-Msie'deen et al. [ASH+13, AHS+14] exists to automatically generate documentation for identified features from source code and corresponding use case diagrams. The authors apply NLP techniques to set existing use case diagrams in relation with the identified features to assign corresponding feature names and descriptions. Thus, the approach allows to increase comprehensibility of features for developers.

In addition, Damevski et al. [DSP16] empirically analyzed how developers manually execute feature location to better understand their approaches and provide improved algorithms to support them. The authors found that additional support for querying source code for features (e.g., through recommendations) is needed and that better integration between tools is needed (i.e., to reduce switching different tools used during the analysis).

Depending on the used techniques, the discussed approaches are able to (semi-)automatically identify source code artifacts belonging to a specific feature. While these approaches in general are capable of identifying features for source code artifacts, they often rely on structures, such as PDGs, that are not available for block-based languages. In contrast, our Matador SPL approach exploits the hierarchical decomposition available in many block-based languages to automatically identify features and corresponding feature models. In addition, it does not rely on execution traces, which would have to be tediously generated for all analyzed variants and corresponding features to apply dynamic feature location techniques.

Feature Location Techniques for Models The approaches most related to our approach for feature location, are approaches analyzing model variants to extract corresponding SPL features.

The approach by Martínez et al. [MZB+15a] applies the notion of interdependent atomic model elements by Ziadi et al. [ZFS+12] and an interdependence relation to identify features (cf. Section 4.8.3). While the approach is able to identify features, the approach heavily relies on the assumption that features are not changed between products. For instance, when copying an existing variant and modifying one of the features for some reason (e.g., to fix bugs), they might be identified as separate features as the atomic model elements changed across the variants. In contrast, our approach

uses a rather simple approach to feature location and exploits the hierarchical decomposition of the analyzed model variants and would identify such fixes to a feature residing in the same hierarchy.

Font et al. [FAH+15, FAH+16a, FAH+16b, FAH+17, Fon17] identified that many automatic variability mining approaches produce results that do not match the expectations of domain experts as they do not involve the expert's knowledge into the variability extraction process. The authors use a mutation-based approach to identify features in a set of input models from a user-provided seed fragment [FAH+15]. Around this seed fragment the algorithm generates mutations conforming to the surrounding elements from the seed model. After pruning the generated mutations using user-defined rules (e.g., excluding mutations containing certain unwanted elements), a set of model patterns is extracted from the initial subset of models. Based on these results, the user is able to select and create a CVL-based variability model with replacement fragments provided by a model library. In continued work, the authors improve these algorithms to enable their application in scenarios where seed fragments cannot be selected precisely (e.g., due to lacking documentation) [FAH+16a]. These algorithms are able to improve over the previous results as the search space is broadened by traversing the complete solution space independent of the initial input variants. In further work, the authors use genetic algorithms combined with IR techniques to not only consider a provided seed fragment, but also a corresponding feature description [FAH+16b, FAH+17]. This way, the authors are capable of iteratively executing the genetic algorithms to, finally, return features that are close to the initial description provided by the user.

The approach by Font et al. [FAH+15, FAH+16a, FAH+16b, FAH+17, Fon17] approaches the identification of variability information from a different angle and tries to use machine learning techniques to automate the identification. The main goal is to reduce the manual effort of domain experts during variability identification. However, this is also a drawback as the experts' only way to influence the variability identification with genetic algorithms is the selection of an initial population and the final assessment of the identified elements. In contrast, our Family Mining and Matador SPL approaches allow experts to define their own metrics and which elements to regard as feature roots. Thus, we allow developers to explicitly influence the way elements are compared and extracted.

Arcega et al. [AFH+15, AFH+16a, AFH+16b, AFH+17] follow a similar approach and apply genetic algorithms to locate features from corresponding traces that were generated at runtime. The authors provide tooling to link source code with corresponding model elements (e.g., the elements used to generate the code) [AFH+16a]. Developers can manually define these relations and the tooling uses them to monitor the artifacts during execution and create corresponding traces. Using such traces, the approach is able to locate features based on a seed fragment (i.e., a user-selected model element) using a mutation-based approach [AFH+15]. In continued work, the authors use IR techniques to allow users to query execution traces for a searched feature [AFH+17].

Similar to the work by Font et al. [FAH+15, FAH+16a, FAH+16b, FAH+17, Fon17], these approaches use machine learning techniques to identify relevant features. While this largely reduced the manual overhead, it also results in limited possibilities on influencing the results of the algorithms. Thus, the user fully relies on the algorithms of the tooling and cannot influence them in cases where undesired results are created. Furthermore, the authors do not describe how the located features can be used to migrate to an SPL. In contrast, our Family Mining and Matador SPL approaches provide means to incorporate domain knowledge in the identification of variability (i.e., using custom metrics and feature extraction rules) and automatically transition to an SPL realization.

Furthermore, different experience reports exist on how to manually transition a set of existing variants from single product development to managed reuse in an SPL. Examples are the reports by Kim et al. [KKK07], Kolb et al. [KMP+05] and Lee et al. [LCK+09]. For instance, Kolb et al. [KMP+05] describe their experience of refactoring the image memory handler of Ricoh's printers into an SPL. While these approaches consider not only the underlying source code, but also set the corresponding design models into relation with identified features, the approaches rely on manual analysis of the refactored systems and only derive corresponding guidelines. In contrast, our approach aims at reducing the manual overhead and automatically transitioning the identified variability and features to an SPL.

Automatic Identification of Cross-Tree Constraints In addition, techniques exist to automatically identify cross-tree constraints from different artifacts. Examples for such identification algorithms are approaches by Nadi et al. [NBK+14], Yi et al. [YZZ+12], Haslinger et al. [HLE13], She et al. [SLB+11, SRA+14] and Martínez et al. [MZB+15b]. For instance, Martínez et al. [MZB+15b] identify cross-tree constraints between the features identified by their Bottom-Up Technologies for Reuse (BUT4Reuse) approach (cf. Section 4.8.3). The authors analyze the structural dependencies between elements to identify requires constraints [MZB+15b]. Furthermore, mutually exclusive constraints are identified by further including information on the maximum number of allowed dependencies per block of elements. All these approaches are generally able to identify sensible cross-tree constraints from existing variants. However, we argue that reverse-engineering such constraints can only show *possible* relations and they have to be manually checked by experts. This is mainly due to the fact that identifying cross-tree constraints for a generated SPL can always only consider the subset of existing variants provided by the user. Thus, generating too restrictive constraints might unnecessarily limit the number of possible variants that can be generated from an SPL.

5.6.3. Automatically Migrating Variability in Software Product Lines

Different approaches exist to migrate identified variability information from compared product variants to an SPL [FTS14]. In general, they can be divided in approaches supporting generation of SPLs for source code and models. As the discussed approaches for identifying variability information in existing product variants (cf. Section 4.8.3) mostly aim at creating corresponding SPL artifacts, we focus our discussion on these approaches and extend them where applicable.

Migrating Variability in Source Code Software Product Lines **Alves et al.** [AMC+05, AGM+06, AMC+07, ACN+08] use a tool-supported strategy to manually refactor existing products into an SPL encoding the variability using AOP (cf., Section 4.8.3). The approach provides refactorings, that can be manually selected by users to extract functionality from the original source code into aspects that can be applied to the core variant of the SPL. The approach does not provide support to automatically extract a corresponding feature model. In contrast, our Matador SPL approach allows to automatically generate an SPL with a feature model allowing configuration of derived variants.

Klatt et al. [KK12, KK13, KKK13, KKS14, KKW14, Kla14] provide developers with detailed variation point models extracted from analyzed product variants (cf. Section 4.8.3). Furthermore, the authors allow definition of SPL profiles that allow structured selection of techniques for the created SPL by users. The generated variation point models are then used to select corresponding variability realization mechanisms, which is supported by automatic recommendations of their SPLevo approach. In addition, the SPLevo approach supports developers in identifying variation points that

contribute to the same feature by analyzing shared terms from their implementations and providing corresponding recommendations. In contrast to the semi-automatic and tool-supported SPL migration by Klatt et al. [KK12, KK13, KKK13, KKS14, KKW14, Kla14], we follow a different strategy. Our Family Mining and the Matador SPL approach allow to include upfront knowledge on the similarity of model elements and, thus, are capable of automatically generating an SPL realization that is close to the original implementation of the migrated product variants. Afterwards, we provide tooling to support developers during refactoring of this SPL towards a solution meeting their problem-specific requirements.

Linsbauer et al. [FLL+14, FLL+15, LFL+15, LLE17] allow extraction of feature-specific code from feature traces (cf. Section 4.8.3). The authors approach follows the goal of supporting developers during clone-and-own development of variants. To this end, the approach provides means to select features and compose the corresponding code to generate variants conforming to dependencies between these features. While this realization is very close to the generation facilities of SPLs, it might require manual adaptations of the composed code (e.g., in cases where the automatic extraction was not able to completely extract features). In contrast, our approach generates a fully functional DOP SPL from existing variants that does not require manual adjustments to the generated variants.

Fenske et al. [FMS+17] provide variant-preserving refactorings to developers, which allow them to align and migrate a number of product variants to an SPL (cf. Section 4.8.3). During these refactorings, the approach also modifies a manually created feature model that initially comprises only the considered products as features and over time is refined to contain the extracted features. While the approach by Fenske et al. focuses on scenarios where the products diverged from each other (e.g., due to renamed methods) and requires tool-supported manual refactorings, our approach assumes that the cloned products are still related enough to be automatically migrated to an SPL. However, using custom metrics allows developers to involve their knowledge about changes between the cloned products in our Family Mining. In addition, applying variant-preserving refactorings similar to those of Fenske et al. prior to the automatic migration to an SPL would also allow our approach to handle such scenarios.

Ziadi et al. [ZFS+12, ZHP+14] describe an approach to identify features in existing source code and migrate them to a FOP SPL (cf. Section 4.8.3). While the generated SPL comprises the artifacts necessary to generate all input variants, the generated feature model is rather simple as it only comprises a single hierarchy with the mandatory core and features with generic names. In contrast, our approach is capable of identifying not only meaningful names from the analyzed model, but also creates an initial hierarchical decomposition of these features based on the model hierarchy.

Nöbauer et al. [NSG14] describe an approach to migrate developed standard software extensions to an SPL (cf. Section 4.8.3). Based on a database, the approach keeps traceability between requirements and source code artifacts and is capable of generating a corresponding feature model. However, in contrast to our approach, this feature model does not comprise restrictions of the variability and by default sets features to be optional.

Reinhartz-Berger et al. [RZW16] describe an ontology-based approach to suggest variability realization mechanisms to developers during the creation of SPLs from existing variants (cf. Section 4.8.3). While their approach supports developers during the manual creation of an SPL, it is not capable of automatically migrating identified variability to an SPL. In contrast, our Matador SPL approach allows developers to automatically derive a concrete SPL realization.

Méndez-Acuña et al. [MGC+16a, MGC+16b] allow automatic extraction of reusable modules from related DSLs (cf. Section 4.8.3). While the encapsulated language modules can be used to create new DSLs by plugging them together, the approach does not use actual SPL generation facilities, such as feature models. In contrast, our Matador SPL approach allows to automatically derive delta modules that can be selected using configurations of a corresponding feature model.

Kästner et al. [KDO14] allow semi-automatic extraction of source code belonging to upfront known features by searching for related source code elements starting from given seed fragments (cf. Section 4.8.3). While Kästner et al. [KDO14] only analyze single product variants to identify features, our Matador SPL approach is able to consider multiple variants and to migrate them with their identified features to a common SPL.

Liu et al. [LBL06] describe a tool-supported approach to refactor single Java applications based on the Java parse tree in Eclipse into FOP SPLs. While the approaches by Kästner et al. [KDO14] and Liu et al. [LBL06] are capable of identifying implementation artifacts for features in single products, they do not automatically generate feature models for the analyzed variants. In contrast, our Matador SPL approach is capable of generating a feature model with linked implementation artifacts in form of delta modules. Furthermore, our approach is not limited to a single product variant, but is able to migrate multiple variants to an SPL.

In addition, **Kästner et al.** [KAK09] describe an approach to transform forth and back between physically separated features (e.g., FOP features) and virtually separated features (e.g., `#ifdefs`). These virtually separated features can be regarded as a 150% representation of variability. Similar to our approach, the authors transform this representation to physically separated modules. However, in contrast, our Matador SPL approach does this based on information extracted from existing product variants, while Kästner et al. can build on variability information of an existing SPL.

Migrating Variability in Model-Based Software Product Lines **Alalfi et al.** [ARS+14] use Matlab/Simulink-specific facilities to encode the identified variability (i.e., the non-cloned parts) in a single model representation (cf. Section 4.8.3). The used Variant Subsystems contain the variability of variation points (e.g., differing blocks) and allow configuration of corresponding variants. Thus, in contrast to our approach, they do not provide classic variant derivation facilities as in SPLs, because the model always contains the complete variability and only certain parts of the model are activated or deactivated. Furthermore, Matlab/Simulink does not provide configuration facilities through high level feature selection and the authors do not generate corresponding feature models.

Zhang et al. [Zha10, ZHM11, ZHM12, Zha14] migrate the identified variability in CVL-based SPLs (cf. Section 4.8.3). By comparing the identified differences of the analyzed model variants, the authors iteratively merge identified variation points in an intermediate CVL model (i.e., the model most common to all other variants is selected as basis) and generate corresponding placement and replacement fragments (i.e., variation points in the base model and corresponding varying model elements). In contrast to this approach, we decided to encode the variability information identified by our Family Mining algorithms in a delta-oriented SPL. While both, the CVL and DOP, allow efficient realization of SPLs, the determining factor for this decision was the higher modularity of delta-oriented SPLs due to their capabilities of creating delta modules. This way, the generated SPL allows developers to react much more flexible to changed requirements as additional delta modules can easily be created (e.g., when realizing new product features in a reactive scenario).

Font et al. [FBH+15] basically rely on the same algorithms as the approach by Zhang et al. [Zha10, ZHM11, ZHM12, Zha14] and only add minor extensions to the variability identification prior to the SPL migration (cf. Section 4.8.3). Thus, the same differences between their solution and our Matador SPL approach apply.

Martínez et al. [Mar16, MZB+15a] encode the variability identified by their MoVaC approach (cf. Section 4.8.3) in a CVL-based SPL using their MoVa2PL tool. For the generated SPLs the authors select a subtractive strategy and generate CVL models comprising all product line features, which are removed during product derivation if necessary. In additional work, the authors use NLP techniques to derive word clouds suggesting possible names for identified features in the analyzed variants [MZB+16]. In contrast to the approach by Martínez et al. [Mar16, MZB+15a], users of our Matador SPL approach are able to select an arbitrary model variant as basis for generated SPLs. As a result, we are able to provide much more flexibility during the transition to an SPL as developers can choose an SPL core that serves their goals most.

Rubin et al. [Rub14, RC12, RC13d] describe different algorithms to compare and merge variability of analyzed product variants (cf. Section 4.8.3). The authors provide a strong theoretical foundation for merging variability relations identified during variability mining of related product variants. Looking at the given details, the authors follow an annotative strategy and annotate for each merged model element the containing variants. However, no concrete details on the actual underlying SPL tooling of their realizations are given. In addition, the features for the approaches in [RC12] are rather coarse grained as only a single feature per variant is introduced and, thus, a fine grained configuration of variants is not possible. In contrast, we provide concrete details on the used techniques and identify much more fine-grained features with corresponding implementation artifacts in the analyzed variants.

Boubakir et al. [BC16] describe a technique that is highly similar to the approach by [RC12] (cf. Section 4.8.3), but does not provide any feature identification. As a result, mostly the same differences compared to our Matador SPL approach apply (i.e., except for the missing feature identification).

Ryssel et al. [Rys14, RPK10, RPK11, RPK12a, RPK12b] use the subsystem model templates merged during the comparison of analyzed model variants and derive corresponding subsystem feature models (cf. Section 4.8.3). These feature models comprise fine-grained features for single blocks or combinations of blocks in the subsystems and, thus, allow detailed configuration of each subsystem in isolation. However, due to the realization as an MSPL comprising the individual subsystem SPLs it is possible to configure complete variants. The authors' SPL realization relies on the FeatureMapper tool that allows creation of feature models and configuration of linked EMF-based models [HKW08]. As the FeatureMapper tool does not support MSPLs, the authors have to execute the derivation for each subsystem template separately to derive complete variants. In contrast, our Matador SPL approach relies on DeltaEcore allowing configuration and derivation of complete variants from the derived feature model in a single step. Furthermore, the feature model derived by our approach does not allow configuration of low-level variability (e.g., selection of single states or transitions) as we argue that such low-level variability should be abstracted by higher level features (e.g., activating multiple of these low-level features).

Sabetzadeh et al. [Sab08, SE03, SE06, SNE+07] do not support migration of compared product variants to an SPL realization as they follow a different goal and aim at merging inconsistent or incomplete views of a system into a single consolidated model (cf. Section 4.8.3).

Nejati et al. [Nej08, NSC+07, NSC+12] merge different viewpoints on the same feature (cf. Section 4.8.3). Thus, the authors initially know which feature the corresponding implementations belong to. As a consequence, the approach does not need to identify features in the analyzed models and only provides a merged model corresponding to an annotative SPL realization of the analyzed feature. In contrast, our approach considers the variability of complete product variants and transfers them to an SPL realization with corresponding features.

Pietsch et al. [PKK+15] exploit the capabilities of their SiLift framework [KKT11, KKO+12, KKT13] to derive delta modules from model variants (cf. Section 4.8.3). While the tool-supported approach is capable of deriving delta modules, application conditions have to be added manually. Furthermore, due to the nature of model differencing algorithms, the generated delta modules store the variability between exactly two models. Thus, additional manual effort is needed to align the delta modules from multiple variants into a common SPL (e.g., by merging them) using the tooling provided by the authors. In contrast, our Matador SPL approach is capable of directly generating delta modules for multiple variants and, additionally, derives a corresponding feature model.

Schwägerl et al. [SW16, SW17a] allow evolution of existing SPLs using a VCS-based system (cf. Section 4.8.3). As developers use this approach to check-out revisions of the SPL through the provided feature model, their approach already has feature information and a corresponding SPL realization available. Thus, it is capable of managing corresponding artifacts and does not need to transition the variability to an SPL from scratch.

In Table 5.4, we summarize all discussed SPL migration approaches and inherit the information on their capabilities of identifying coarse-grained and fine-grained variability for compared variants as well as their adaptability for new languages and adjustability to user-defined settings from Table 4.8. In addition to this information, we extend the table with details on the capabilities of the discussed approach to migrate existing variants to an SPL and whether they allow derivation of features with corresponding feature models to configure the SPL's variants.

5.6.4. Restructuring Software Product Lines

In different areas of software development, controlled evolution of artifacts plays an important role to successful adaptation of existing artifacts for new requirements. Thus, research exists to support developers in this work and even provide semi-automated operations to execute consistent refactorings (e.g., by keeping mappings between artifacts aligned). In this section, we concentrate on refactoring approaches in the context of SPLs [FTS14], which are most related to our approach.

First of all, work on analyzing and understanding evolution of SPLs exists. Examples are the approaches by Bürdek et al. [BKL+16], Dintzner et al. [DDP18] and Passos et al. [PTD+16]. For instance, Bürdek et al. [BKL+16] apply the SiLift approach by Kehrer et al. [KKT11] to automatically calculate the semantic differences between compared feature models and classify the identified corresponding edit operations. In addition, Dintzner et al. [DDP18] use their tool Feature EVolution ExtractoR (FEVER) [DDP16] to automatically extract the evolution operations that were applied for source code using pre-processor directives from version control histories. Based on a exploratory study, the authors analyze the co-evolution of different artifacts of an SPL to understand how often developers are faced with such challenges. Furthermore, Passos et al. [PCA+13, PTD+16] identify and describe patterns of co-evolution in SPLs. In contrast to these approaches, our Matador SPL actively supports developers during evolution of SPLs automatically derived from input variants.

		Coarse-Grained Variability	Fine-Grained Variability	SPL Migration	Feature Model	Adaptability	Adjustability
Family Mining	M	+	+	+	+	+	+
Alalfi et al. [ARS+14]	M	–	+	–	–	–	–
Alves et al. [AMC+05]	SC	–	+	o semi-automatic	–	o no tooling	+
Boubakir et al. [BC16]	M	–	o not explicit	+	–	o no tooling	–
Fenske et al. [FMS+17]	SC	–	–	o semi-automatic	+	+	+
Font et al. [Fon17]	M	–	+	+	+	+	o EMFCompare
Kästner et al. [KDO14]	SC	–	–	o single products	–	+	+
Kästner et al. [KAK09]	SC	–	–	o input	–	–	–
Klatt et al. [Kla14]	SC	–	+	o semi-automatic	+	+	+
Linsbauer et al. [LLE17]	B	–	–	o similar to SPLs	+	+	–
Liu et al. [LBL06]	SC	–	–	+	–	+	+
Martínez et al. [Mar16]	B	–	–	+	+	o no tooling	+
Méndez-Acuña et al. [MGC+16b]	SC	–	o only modules	o similar to SPLs	–	–	–
Nejati et al. [Nej08]	M	–	o not explicit	+	o input	–	+
Nöbauer et al. [NSG14]	SC	–	–	+	+	+	–
Pietsch et al. [PKK+15]	M	–	o not explicit	+	–	+	–
Reinhartz-Berg et al. [RZW16]	SC	o variation points	o only partial	–	–	o no tooling	–
Rubin et al. [Rub14]	M	–	o not explicit	+	o coarse	o no tooling	o only [RC12]
Ryssel et al. [Rys14]	M	o for subsystems	+	+	+	–	+
Sabetzadeh et al. [Sab08]	M	–	o not explicit	–	–	o no tooling	–
Schwägerl et al. [SW16]	M	–	–	o input	o input	+	–
Zhang et al. [Zha14]	M	–	+	+	+	+	o EMFCompare
Ziadi et al. [ZHP+14]	SC	–	–	+	+	+	–

Supported artifacts: *M* = models *SC* = source code *B* = both, models *and* source code

Table 5.4.: Comparison of our Matador SPL approach in Chapter 5 with related approaches.

Research specifically targeting refactoring of SPLs concentrates on defining consistent refactoring operators to support developers during evolution of existing SPL artifacts. Examples are the approaches by Alves et al. [AGM+06], Thüm et al. [TBK09], Schulze et al. [STK+12, SRS13] and Seidl et al. [SHA12]. Alves et al. [AGM+06] introduce basic operations for unidirectional and bidirectional refactorings to feature models and provide an approach for showing the soundness of these operations. Furthermore, Thüm et al. [TBK09] define an algorithm to analyze the relations between two feature models and to reason about the applied changes with respect to the valid configurations of the corresponding SPL. While the previously discussed refactorings concentrate on edit operations for feature models, they do not consider other artifacts (e.g., to preserve the behavior of the SPLs' variants defined by configurations). In contrast, Schulze et al. [STK+12, SRS13] describe behavior-preserving refactoring approaches for FOP and DOP SPLs with corresponding configurations. For instance, the authors introduce refactorings for DOP SPLs and code-smells (e.g., undesired programming patterns) to guide developers when applying these refactorings [SRS13]. The proposed approach considers all artifacts relevant for DOP SPLs (i.e., delta modules, mappings, application order, feature model and constraints). To this end, the authors define different refactorings that allow consistent refactoring of these artifacts (e.g., renaming of artifacts, extraction of delta actions, merging of delta modules). In addition, Seidl et al. [SHA12] describe operations for co-evolving the mappings between implementation artifacts and features during refactorings.

The discussed approaches define concrete operations for refactoring SPLs while preserving their behavior during these edit operations. The operators provided by our Matador SPL approach are an implementation of these refactorings for our specific setting of supporting developers during the migration from existing variants to a corresponding SPL. Thus, the operators are implemented to refactor an automatically derived initial SPL, which is close to the existing single variant implementations, into an SPL solution meeting the requirements of experts.

5.7. Chapter Summary

Manually encoding variability information identified for a set of model variants in a corresponding SPL is a tedious and time-consuming task. Using our Matador SPL approach developers are able to automatically transition variability information identified by our Family Mining approach (cf. Chapter 4) to a delta-oriented SPL. From 150% models identified by the Family Mining, our approach allows to automatically derive a delta language that is specifically tailored towards the used modeling language (cf. Section 5.2). Using this delta language, we are able to automatically encode the variability information in a delta-oriented SPL (cf. Section 5.3 and Section 5.4). The generated SPL artifacts comprise features with corresponding delta modules and a feature model allowing configuration of products that should be derived from the SPL. These artifacts can be refactored towards the users' requirements using the refactoring operators provided by our corresponding tooling (cf. Section 5.5). Overall, the migration effort towards an SPL is largely reduced as the identified variability information is automatically encoded in artifacts. Furthermore, the created SPL provides a direct benefit as it not only allows generation of the input variants, but also enables users to configure new products using the derived feature implementations. Thus, the Matador SPL approach reduces the barrier of migrating existing model variants to an SPL as it is capable of completely automating this migration step towards managed reuse of functionality.

Part III.

Realization and Application

6 Realization as the Family Mining Framework

The concepts for variability analysis on the coarse-grained and fine-grained level of model variants as well as their migration to a delta-oriented SPL presented in this thesis are realized in the Family Mining framework. The Family Mining framework provides essential data structures and tooling that can serve as a basis for a detailed variability analysis of model variants. The framework provides import and export functionality for models and allows easy analysis of their contents to identify variability relations. The framework relies on the model-based techniques of the EMF and allows easy processing of models using an Ecore-based meta-model.

The Corevid approach (cf. Chapter 3) allows analysis of their coarse-grained relations to detect clusters and outliers using the Samos framework by Babur et al. [Bab16, BCB16, BCV+16, BC17, BCB18]. Based on theses results, it is possible to apply our Family Mining to identify more fine-grained details of the analyzed models. Using the provided facilities of our Family Mining framework, it is possible to either manually or semi-automatically adapt the generic concepts of this thesis for fine-grained variability analysis of new block-based languages (cf. Chapter 4). For this purpose, the framework additionally provides the Vampire DSL (cf. Section 4.2.4, Section 4.3.3 and Section 4.6.1) to automate large parts of the adaptation process and to allow easy integration of new languages for variability analysis. The identified fine-grained relations can be used to automatically migrate to a delta-oriented SPL using our Matador SPL approach. The Matador SPL approach (cf. Chapter 5) provides tooling to manually refactor the derived SPL using corresponding operators ensuring the integrity of the SPL artifacts.

Chapter Outline The focus of this thesis is to provide algorithms and tooling that can easily be reused and adapted for the variability analysis in different settings and of different languages. Thus, we focus on giving corresponding details during the description of our realization by highlighting parts that are *core* to the realization (i.e., parts provided by us), parts that can be *automatically generated* and parts that have to be *user-provided*. In this context, we also discuss which stakeholders should be involved in the steps of adapting the concepts for new languages. In the following sections, we explain the realization details of each framework part:

- *Section 6.1:* In this section, we give an overview of the Samos framework and how the Corevid approach (cf. Chapter 3) extends the corresponding facilities to allow easy analysis of coarse-grained relations between model variants. Based on these details, we explain how developers can easily provide extensions that allow analysis of new languages.
- *Section 6.2:* In this section, we give an overview of the Family Mining framework and the tooling provided for an easy extension of the framework. Starting with the general idea of the framework, we explain how it executes concrete comparisons using different phases and

how corresponding algorithms can be provided to the framework by the user (e.g., to adapt the algorithms for new languages).

- *Section 6.3:* In this section, we explain the realization of the Vampire DSL (cf. Section 4.2.4, Section 4.3.3 and Section 4.6.1). Furthermore, we give details on how users can use the DSL's facilities to derive concrete adaptations of the Family Mining algorithms for a fine-grained variability analysis of models in new languages.
- *Section 6.4:* In this section, we give details on the realization of the Matador SPL approach (cf. Chapter 5) and how it is integrated in the Family Mining framework for an easy migration of identified variability relations to a delta-oriented SPL.

6.1. Cluster and Outlier Detection using the Corevid Approach

As mentioned our Corevid approach relies on the Samos framework by Babur et al. [Bab16, BCB16, BCV+16, BC17, BCB18] (cf. Chapter 3). Although, the Samos framework relies on the facilities of the Eclipse *Rich Client Platform (RCP)* and the EMF, it is not realized as an explicit Eclipse plug-in architecture and rather uses standard object-oriented techniques. The concrete algorithms are implemented partly in Java and partly in R[1]. R is an open-source environment for statistical computations and visualization of corresponding results. To be more precise, all techniques concerning the IR-feature extraction and the comparisons are realized in Java and all algorithms concerning the clustering are realized in R.

6.1.1. General Realization of the Corevid Approach

In Figure 6.1, we present the component diagram for our Corevid approach. In this case, the components should be rather interpreted as conceptually distinct and modular steps of the workflow in the Samos framework than single cohesive components. We highlight core parts in *blue* with a *core* pictogram, user-provided parts in *orange* with a *hand* symbol and automatically generated parts in *green* with *gears*. In Figure 6.1, all core parts are provided by the Samos framework by Babur et al. [Bab16, BCB16, BCV+16, BC17, BCB18] except for the *Base Meta-Model Provider*, which is part of the Family Mining framework (cf. Section 6.2). However, these parts also include the extensions for our Corevid approach that we directly implemented in the Samos framework (e.g., the tokenization of *snake case* identifiers or the *Bray-Curtis Dissimilarity*). The user-provided parts consist of a) a *Language-Specific Importer* and b) a corresponding *Extraction Scheme & Parameter Provider*. This component configures the Samos framework to extract sensible n-grams and to process them during the clustering (e.g., by using the necessary NLP options and comparison of the n-grams). The *Language-Specific Importer* can be reused from the adaptation of the Family Mining framework as it is also required to process models of a new language for the corresponding fine-grained variability analysis (cf. Section 6.2). The remaining *Language-Specific Meta-Model Provider* can be automatically generated using the Vampire DSL (cf. Section 6.3) of the Family Mining framework.

The following list gives a detailed overview of all components together with their dependencies to other components where applicable:

[1] https://www.r-project.org/

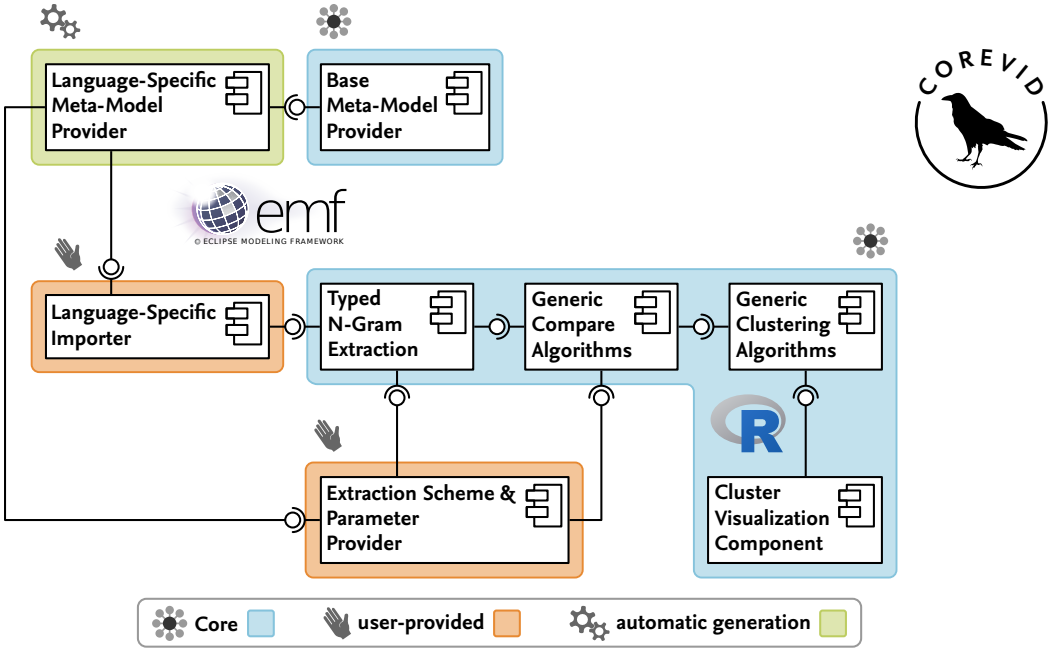


Figure 6.1.: Components of the Corevid approach.

- *Base Meta-Model Provider:*
  - *Description:* This component provides the base meta-model, which can be used as the core of meta-models for our generic Family Mining. Thus, it is contained in the core of the Family Mining framework. We leave a detailed discussion to Section 6.2.
- *Language-Specific Meta-Model Provider:*
  - *Description:* This component provides a language-specific meta-model, which can be generated using the Vampire DSL of the Family Mining framework. We leave a detailed discussion to Section 6.2 and Section 6.3.
  - *Requires:*
    1. *Base Meta-Model Provider:* This requirement is optional as meta-models only rely on the base meta-model when using corresponding parts.
- *Language-Specific Importer:*
  - *Description:* This component provides the importer to translate a model from its original modeling notation to an instance of a corresponding meta-model. This component is also used for the Family Mining. We leave a detailed discussion to Section 6.2. This importer can be replaced by a generic EMF importer, also provided by the Family Mining framework, if the analyzed models were stored as XMI files.
  - *Requires:*
    1. *Language-Specific Meta-Model Provider:* Required as this component provides the necessary meta-model description used for instances of imported models.

- *Extraction Scheme & Parameter Provider:*
  - *Description:* This component provides a language-specific extraction scheme for the analyzed modeling language together with settings controlling the extraction (i.e., the size of extracted n-grams) and the comparison of n-grams (i.e., applied NLP techniques and used weighting schemes).
  - *Requires:*
    1. *Language-Specific Meta-Model Provider:* Required as it provides the meta-model during import of models and, thus, is needed for language-specific extraction schemes.
- *Typed N-Gram Extraction:*
  - *Description:* This component extracts typed n-grams from input models using the EMF API together with the user-specified extraction scheme.
  - *Requires:*
    1. *Language-Specific Importer:* Required as it imports models from tool-specific notations to the meta-model and, thus, provides the models for the n-gram extraction.
    2. *Extraction Scheme & Parameter Provider:* Required as it provides the language-specific extraction scheme, which controls the size of the typed n-grams (i.e., the used n) and how the model details are encoded in the n-grams.
- *Generic Comparison Algorithms:*
  - *Description:* This component provides the comparison algorithms of the Samos framework with its IR and NLP capabilities.
  - *Requires:*
    1. *Typed N-Gram Extraction:* Required as it provides the typed n-grams that are processed by this component prior to the clustering.
    2. *Extraction Scheme & Parameter Provider:* Required as it provides the settings for the executed comparisons (i.e., applied NLP techniques and used weighting schemes).
- *Generic Clustering Algorithms:*
  - *Description:* This component provides the clustering algorithms of the Samos framework and uses the R packages `hclust` for HAC and `vegan` for *Bray-Curtis Dissimilarity* in its concrete realization.
  - *Requires:*
    1. *Generic Comparison Algorithms:* Required as it provides the comparisons results for the typed n-grams that should be used to cluster the models.
    2. *Cluster Visualization Component:* Required as it provides the functionality to generate dendrograms for the clustering results.
- *Cluster Visualization Component:*
  - *Description:* This component provides the dendrogram visualization of the clustering results for the Samos framework and uses the capabilities of R for the concrete realization.

Using the described architecture in Figure 6.1, it is particularly easy to adapt the cluster and outlier detection provided by the Corevid approach for new languages. Large parts of the realization can be reused through the Samos framework and the extensions by the Corevid approach (e.g., the improved processing of identifiers). For instance, all concrete algorithms are already implemented, and the larger part of settings can be reused (e.g., the executed comparison and clustering algorithms). Furthermore, the language-specific meta-model and a corresponding importer also have to be realized for the execution of fine-grained variability analysis in the Family Mining and can easily be realized using the corresponding tooling (cf. Section 6.2 and Section 6.3). Thus, only small language-specific parts have to be manually implemented (i.e., a corresponding extraction scheme) and only a small set of problem-specific settings have to be found. For example, although, the default value recommended for n-gram size is $n = 2$ following the evaluation by Babur et al. [BC17], it can be replaced by a custom value in cases where a different n would yield better results (cf. [BC17]).

In addition, it is advisable to check and adjust the NLP processing capabilities of the Samos framework prior to executing the comparison of extracted typed n-grams. For example, depending on the used language, users might find that an extension of the applied stop word list is necessary or a different option for preprocessing identifiers in *camel case* (e.g., `token1Token2Token3...`) is needed. As a result, the required knowledge to set up the Corevid approach is limited to a small number of settings that are available through knowledge of the used tool-specific notation and corresponding company-specific implementation guidelines. In case this knowledge is not available, we refer to Section 4.1 for details on how to execute a sensible analysis of available language concepts and their usage in the company. For the evaluation of this thesis, we adapted the Corevid approach for the coarse-grained variability analysis of statechart variants using the details discussed in Chapter 3.

6.1.2. Adapting and Applying the Corevid Approach

In Figure 6.2, we show the workflow on how to adapt the Corevid approach for a new language or different settings (e.g., when applying it in a different project) and apply it to corresponding models. We focus on the stakeholders involved and clearly outline their different roles.

For the adaptation of the Corevid approach for new languages or changed settings (e.g., when analyzing a project with completely different settings), we assume that a *knowledgeable developer* is available. This user should have basic knowledge of meta-modeling, the used modeling language and the product family that should be analyzed with the Corevid approach. This developer is most likely a senior developer with experience and a general overview of project goals in the analyzed model variants. Prior to actually adapting the Corevid approach, this developer has to define a language-specific meta-model and implement a corresponding importer following the guidelines in Section 4.2 or using the Vampire DSL to automatically generate the meta-model (thus, we marked it in Figure 6.2 with an asterisk). Based on this meta-model, the developer can define an extraction scheme to extract IR-features in form of typed n-grams for the analyzed modeling language. While we recommend using typed bigrams (i.e., with $n = 2$), it can have positive effect to use larger n-grams [BC17]. Thus, an optional step can be to experiment with the size (i.e., the n) of the typed n-grams. Afterwards, the comparison settings to analyze the n-grams and an appropriate distance measure have to be selected. Here, we recommend the settings described in Chapter 3 and only language-specific or project-specific settings for the executed NLP have to be selected (e.g., specific processing of identifiers or extensions of the stop word list).

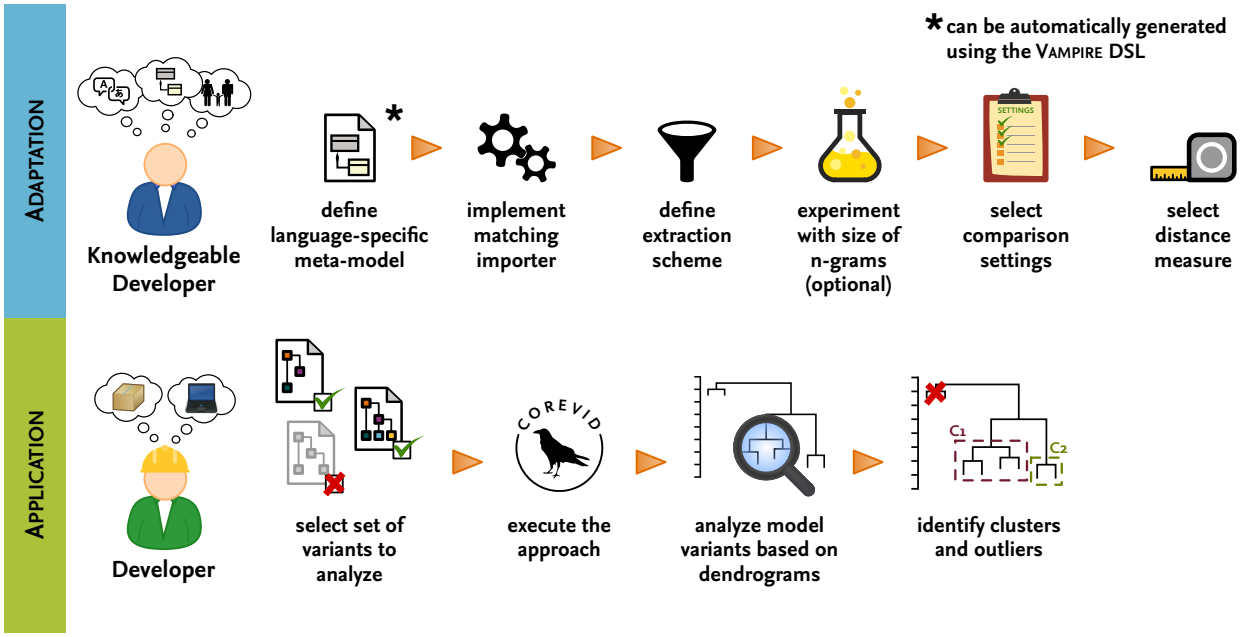


Figure 6.2.: Workflows to adapt and apply the Corevid approach for new languages and different settings.

Using the adapted Corevid approach, any *developer* with at least basic knowledge of the developed products is able to analyze user-selected model variants in the corresponding modeling language by generating corresponding dendrograms. These developer should at least have general knowledge of the used modeling language and developed products to be able to evaluate the generated dendrogram during an exploration of the suggested clusters and outliers. Otherwise, the dendrograms might suggest valid results, but the developers are not able to properly interpret the results to identify sensible clusters and remove outliers.

6.2. Custom-Tailored Variability Mining using the Family Mining Framework

The Family Mining framework provides a general execution framework to configure the execution of variability analysis algorithms. During its implementation, we put emphasis on providing generic data structures and concepts related to variability mining, not only in the context of our Family Mining algorithms. The framework comprises generic facilities to implement the different steps involved in variability mining with data structures (e.g., to store intermediate results) supporting them and possibilities to realize a highly configurable system (e.g., allowing the configuration of metrics through the framework's GUI). The complete Family Mining framework is realized using Java and relies on the Eclipse RCP using the EMF for defining meta-models.

6.2.1. General Realization of the Family Mining Framework

In Figure 6.3, we show all steps that are executed when triggering the Family Mining framework. These steps can be thought of as a pipeline that is processing a set of input models to generate the desired result. Each box in Figure 6.3 refers to an extension point allowing definition of corresponding algorithms.

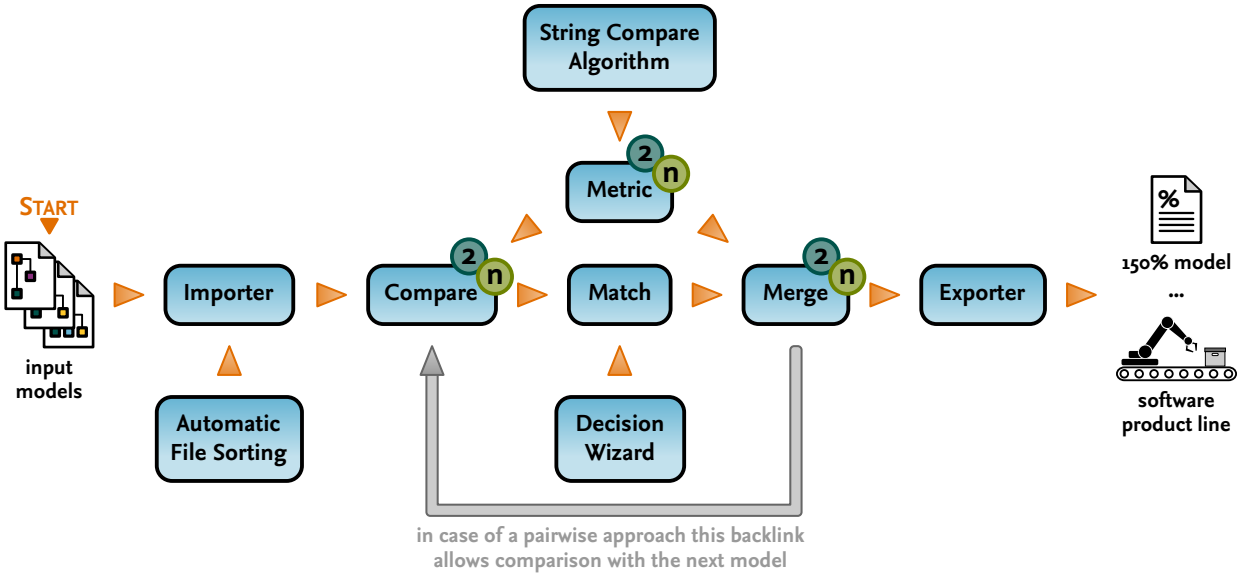


Figure 6.3.: Steps of the processing pipeline in the Family Mining framework.

The framework in general allows to select two processing options for input data. First, it provides *pairwise* processing, which is used by the Family Mining algorithms used in this thesis (cf. Chapter 4). This approach uses an iterative strategy and processes the input models based on a predefined order (either user-defined or automatically derived through the *Automatic File Sorting* extension point). After importing the first two models, the corresponding meta-model instances are compared, matched and their variability is merged using selected algorithms. At this point, the results are either exported (if no remaining models exist) or compared with the next on demand imported model. Second, the framework provides the possibility to implement *n-way* processing of models. For these algorithms, all input models are directly imported and, afterwards, compared, matched and merged in a single iteration that exports the results at the end. Steps that provide different extension points based on the selected approach are marked with a 2 for an iterative pairwise processing and an *n* for n-way processing. For a detailed discussion of the extension points, we refer to Appendix F, where we provide further details on their interfaces, realization, supporting data structures and corresponding default implementations.

As the extension points shown in Figure 6.3 seem to be fairly close to the structure of our Family Mining algorithms, one can argue about the generality of the framework. However, in research directly or indirectly related with this thesis, we demonstrated that the framework can successfully be used as a basis and extended through its extension points for different variability analyses having completely other requirements than our Family Mining. For example, in [SWC+17], we propose and demonstrate the Reverse Signal Propagation Analysis (RSPA) that identifies variation points between Matlab/Simulink models by encoding their graphs (i.e., their blocks and connectors) based on the Matlab/Simulink-specific unique identifiers. This approach captures evolution between Matlab/Simulink models by identifying executed changes and aggregating them in cohesive variation points. Furthermore, in [WWS+17c], we propose and demonstrate variability mining for high-level descriptions of technical architectures (i.e., complete systems comprising software and hardware components). This approach provides developers with information on common and

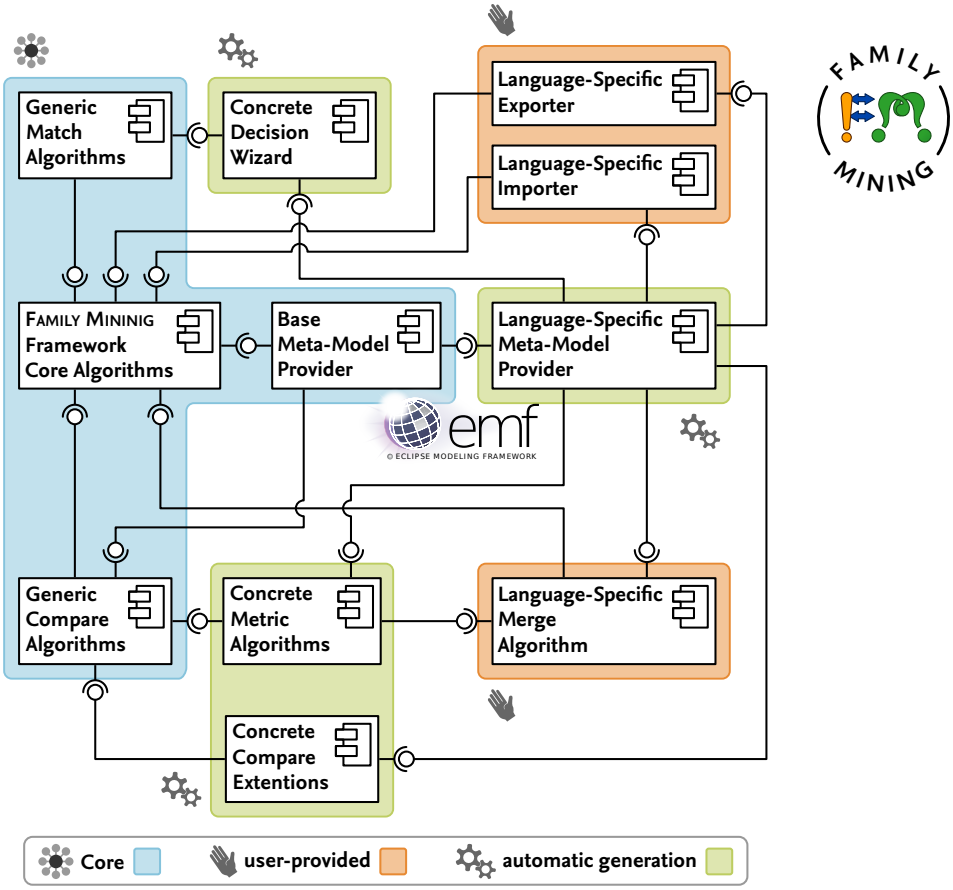


Figure 6.4.: Components of the Family Mining framework.

varying parts of such systems and, thus, allows detailed analysis of reuse potential across the systems to, for example, remove unnecessary variability (e.g., redundant solutions). In both cases, the Family Mining framework and its provided infrastructures played essential roles as they allowed reuse of existing functionality (e.g., the importer for Matlab/Simulink models) and easy extension capabilities (e.g., implementation of new comparison algorithms for technical architectures).

In Figure 6.4, we show the component diagram for our Family Mining framework with corresponding details for the concrete Family Mining algorithms discussed in this thesis. The different components refer to plugs-ins extending the Family Mining framework using the corresponding extension points shown in Figure 6.3 to integrate custom functionality in the framework. We highlight core parts in *blue* with a *core* pictogram, user-provided parts in *orange* with a *hand* symbol and automatically generated parts in *green* with *gears*. The core components provide the general framework functionality to configure the processing pipeline and select different algorithms for each step in the pipeline. In addition, the *Generic Comparison Algorithms* and *Generic Match Algorithms* as well as the *Base Meta-Model Provider* are contained in the core implementation to allow easy reuse for new languages. The user-specified components comprise *Language-Specific Importers*, *Exporters* and *Merge algorithms* for the analyzed language. The remaining *Language-Specific Meta-Model Provider*, *Concrete Decision Wizard*, *Concrete Metric Algorithms* and *Concrete Compare Extensions* can be automatically generated using the Vampire DSL (cf. Section 6.3) of the Family Mining framework.

The following list gives a detailed overview of all components together with their dependencies to other components where applicable:

- *Base Meta-Model Provider:*
  - *Description:* This component provides the base meta-model, which can be used as the core of meta-models for our generic Family Mining. Thus, it is contained in the core of the Family Mining framework and provides the facilities for creating meta-models enabling our generic Family Mining algorithms discussed in Chapter 4.
- *Language-Specific Meta-Model Provider:*
  - *Description:* This component provides a language-specific meta-model, which can be generated using the Vampire DSL (cf. Section 6.3) and, thus, allows easy adaptation of new languages for our Family Mining.
  - *Requires:*
    1. *Base Meta-Model Provider:* This requirement is optional as meta-models only rely on the base meta-model when using corresponding parts. Besides, it is also possible to use corresponding `EAnnotations` instead.
- *Family Mining Framework Core Algorithms:*
  - *Description:* This component provides facilities to configure the Family Mining pipeline and interfaces to extend existing core algorithms with additional functionality.
  - *Requires:*
    1. *Base Meta-Model Provider:* Required to allow usage of meta-models relying on the base meta-model.
    2. *Language-Specific Importer:* Required to allow import of models into instances of a language-specific meta-model.
    3. *Generic Comparison Algorithms:* Required to execute the generic comparison algorithms discussed in Section 4.4.
    4. *Generic Match Algorithms:* Required to execute the generic match algorithms discussed in Section 4.5.
    5. *Language-Specific Merge Algorithm:* Required to merge the variability information identified for model variants into a 150% model representation.
    6. *Language-Specific Exporter:* Required to allow export of the merged 150% model to a language-specific representation.
- *Language-Specific Importer:*
  - *Description:* This user-specified component provides algorithms to import notation-specific models into a corresponding meta-model representation.
  - *Requires:*
    1. *Language-Specific Meta-Model Provider:* Required as this component provides the necessary meta-model description used for instances of imported models.

- *Generic Comparison Algorithms:*
  - *Description:* This component provides the generic comparison algorithms, which are executed on the level of the base meta-model representation (i.e., nodes and edges) of the analyzed models (cf. Section 4.4).
  - *Requires:*
    1. *Base Meta-Model Provider:* Required as the component executes the general comparisons of models on the level of the base meta-model classes or corresponding `EAnnotations`.
    2. *Concrete Compare Extensions:* Required as this component provides extensions to identify information required by the comparison algorithm, which is not available on the generic level of the base meta-model (e.g., the execution start nodes).
    3. *Concrete Metric Algorithms:* Required as this component provides the detailed comparison logic that is executed for low-level model elements (e.g., when comparing states or transitions in statecharts).
- *Concrete Compare Extensions:*
  - *Description:* This component provides additional user-selected algorithms that are required by the generic comparison algorithm to identify, for example, the execution start in models (e.g., *Inport Blocks* in Matlab/Simulink) or subsequent model elements for the comparison (e.g., outgoing transitions for states of statecharts).
  - *Requires:*
    1. *Language-Specific Meta-Model Provider:* Required as this component provides the necessary meta-model description used for instances of imported models and, thus, allows additional analysis of model details.
- *Concrete Metric Algorithms:*
  - *Description:* This component provides user-selected concrete algorithms to compare low-level model elements on a detailed level (e.g., considering their attributes and neighborhood) and to calculate their similarity.
  - *Requires:*
    1. *Language-Specific Meta-Model Provider:* Required as this component provides the necessary meta-model description used for instances of imported models and, thus, allows detailed comparison of model elements.
- *Generic Match Algorithms:*
  - *Description:* This component provides the generic match algorithms, which are executed to identify distinct relations between the elements of models from potentially ambiguous results of the initial comparison algorithms (cf. Section 4.5).
  - *Requires:*
    1. *Concrete Decision Wizard:* Required to solve situations where multiple matching options exist and no direct solution can be found by the matching (cf. Section 4.5).

- *Concrete Decision Wizard:*
  - *Description:* This component provides user-selected algorithms to determine distinct relations between model elements whose relations cannot be unambiguously identified using the generic matching algorithm (i.e., a conflict prevents matching them).
  - *Requires:*
    1. *Language-Specific Meta-Model Provider:* Required as this component provides the necessary meta-model description used for instances of imported models and, thus, allows additional analysis of relations in conflicting situations.
- *Language-Specific Merge Algorithm:*
  - *Description:* This user-provided component implements concrete language-specific algorithms to merge a 150% model for compared models (cf. Section 4.6).
  - *Requires:*
    1. *Concrete Metric Algorithms:* Required as the concrete metric provides thresholds for the categorization of relations into mandatory, alternative and optional elements.
    2. *Language-Specific Meta-Model Provider:* Required to store the merged models in a notation conforming to the language-specific meta-model.
- *Language-Specific Exporter:*
  - *Description:* This user-specified component provides algorithms to export meta-model representations of merged 150% models in a language-specific notation (e.g., their original notation). In addition, a generic exporter is provided to export an Ecore representation of the merged 150% model in form of EMF XMI files.
  - *Requires:*
    1. *Language-Specific Meta-Model Provider:* Required as this component provides the necessary meta-model description used for instances of imported models.

Using the described architecture, it is easily possible to instantiate our Family Mining framework for new languages. Most of the algorithms are provided through the core components of the Family Mining framework, and large parts of the extensions required for new languages can be generated using the Vampire DSL (cf. Section 6.3). Furthermore, all user-provided components concern parts, where an automatic generation of corresponding algorithms is not possible or even not desirable. In case of the importers and exporters, the generation of corresponding functionality is not possible as mappings have to be created between language elements and their representation in the used meta-model. Furthermore, we even argue that a manual implementation is desirable to keep full control over the representation of imported or exported information in the created models. Similarly, the language-specific merge algorithms can only be implemented by users to control the correct representation of the identified variability relations. Overall, the manual effort for the implementation of such algorithms is reasonable, because it is limited to model-to-model transformations between the imported models and their meta-model representation. Furthermore, the merging of variability information follows basically the same principals, but is only extended with additional annotations regarding the identified variability, which is automatically categorized

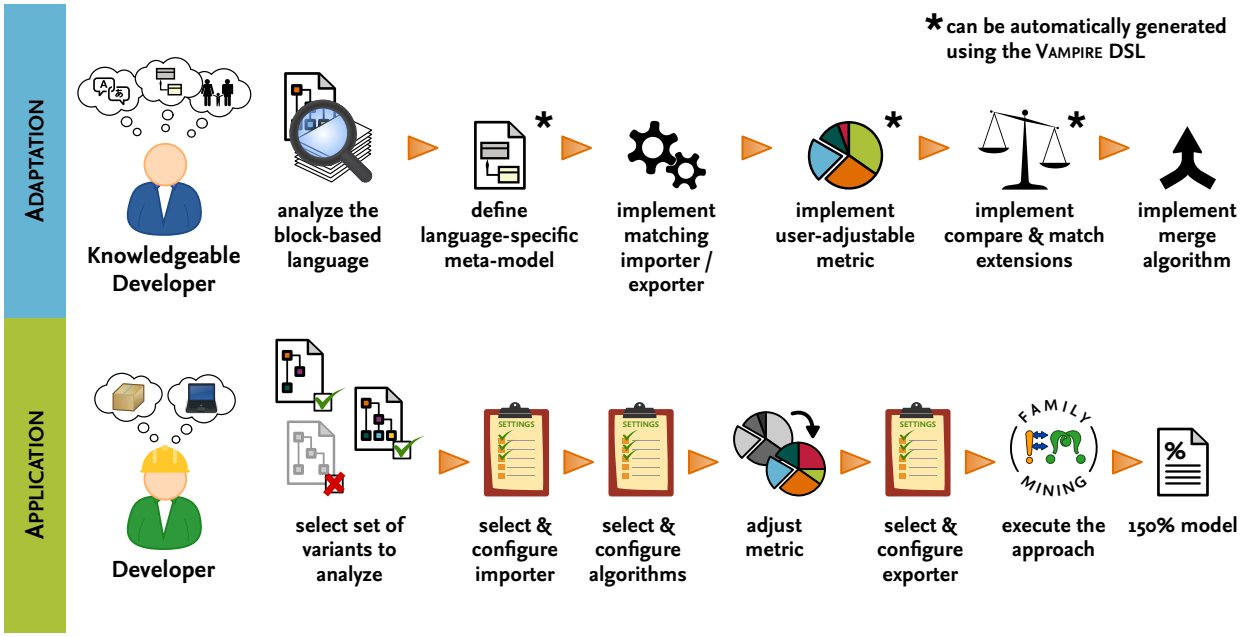


Figure 6.5.: Workflows to adapt and apply Family Mining for new languages and different settings.

based on the generated metric thresholds. For the evaluation of this thesis, we adapted the Family Mining algorithms for the fine-grained variability analysis of statecharts and Matlab/Simulink variants using the details discussed in Chapter 4.

6.2.2. Adapting and Applying Variability Mining in the Family Mining Framework

In Figure 6.5, we show the workflow on how to adapt the Family Mining approach for a new language or different settings (e.g., when applying it in a different project) and apply it to corresponding models. For the adaptation of the Family Mining approach, we assume that a *knowledgeable developer* is available. This developer should have basic knowledge of meta-modeling, the used modeling language and the product family that should be analyzed with the Family Mining approach. This developer is most likely a senior developer with experience and a general overview of project goals in the analyzed model variants. The general adaptation follows the workflow in Chapter 4. The parts marked with an asterisk can be generated based on Vampire DSL descriptions (cf. Section 6.3).

Using the adapted Family Mining approach, any *developer* with at least basic knowledge of the developed products is able to analyze user-selected model variants by deriving fine-grained variability relations in form of 150% models for the corresponding modeling language. These developers should have at least general knowledge of the used modeling language and developed products to be able to evaluate the identified variability results and adjust the provided metrics. After selecting a set of variants through the context menu in Eclipse (cf. Figure 6.6), the developer selects a suitable importer for these files in the opened Family Mining wizard (cf. Figure 6.7). In the subsequent steps of the Family Mining wizard, the developer can select and configure the algorithms that should be applied to these models. These selection options follow the pipeline in Figure 6.3 (i.e., for each extension point a corresponding algorithm has to be selected). Afterwards, it is possible to adjust the weights and thresholds of the selected metric through a corresponding dialog (cf. Figure 6.8)

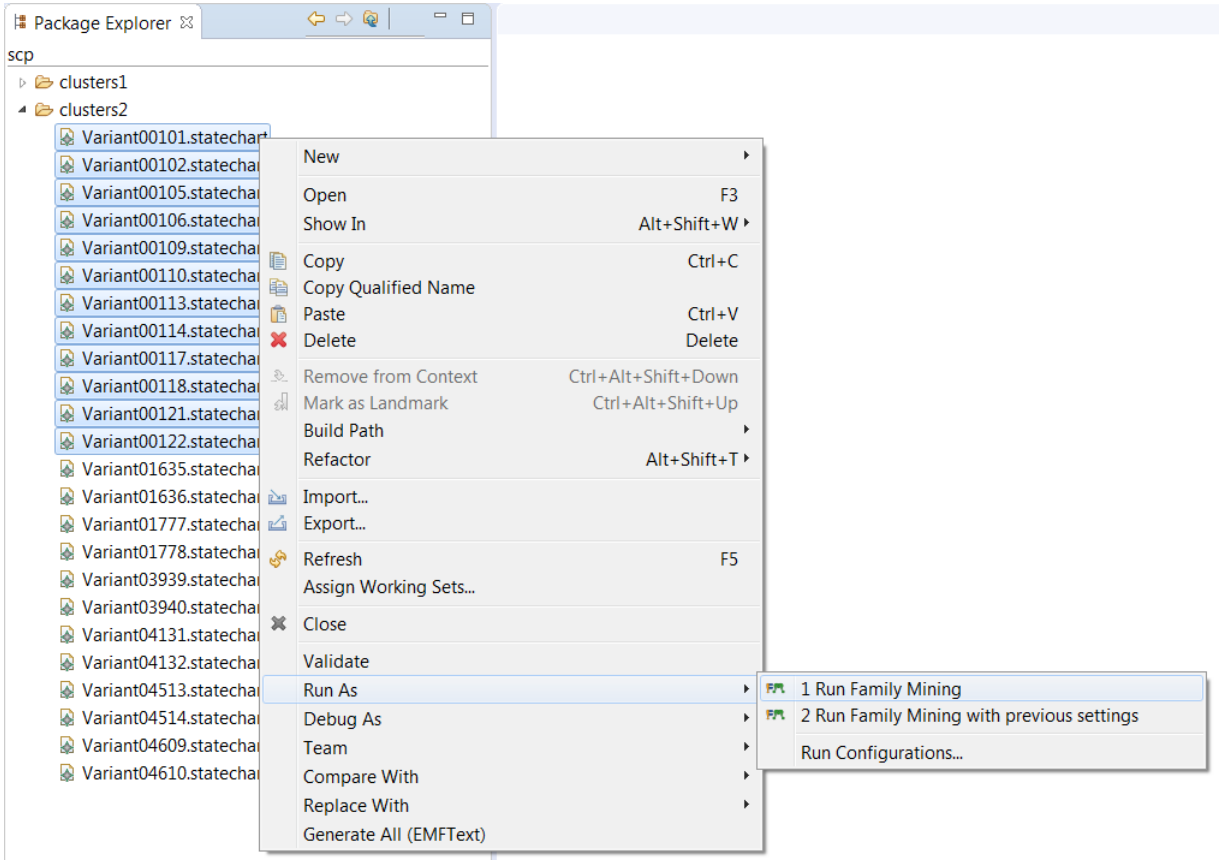


Figure 6.6.: Context menu to execute Family Mining for a selected set of model variants.

and select an exporter to export the results (e.g., a 150% model as in Figure 6.5 or a delta-oriented SPL as described in Section 6.4). Based on the created configuration it is possible to actually execute the Family Mining approach. The last configuration for an execution is stored to allow users to apply it again in a subsequent execution (cf. the second menu entry in Figure 6.6).

6.3. Adapting Custom-Tailored Variability Mining using the Vampire DSL

To ease adaptation of the Family Mining algorithms for new block-based languages, we realized the Vampire DSL as additional support for developers. As discussed in Chapter 4, the general idea is to allow developers to define meta-models together with corresponding metrics to make the generic Family Mining comparison algorithms available more easily for their language. The realization of the Vampire DSL uses Java and relies on the Eclipse RCP and the facilities of the EMF. Furthermore, we use the Xtext framework to develop the Vampire DSL using the corresponding grammar language. The underlying structures for languages developed with Xtext are based on the Ecore meta-model and allow automatic derivation of parsing facilities for the developed textual language with corresponding *Integrated Development Environment (IDE)* infrastructures. Files written in a developed DSL are parsed by Xtext and translated to model-based representations conforming to the defined grammar and, thus, allow processing of textual files using model-based techniques.

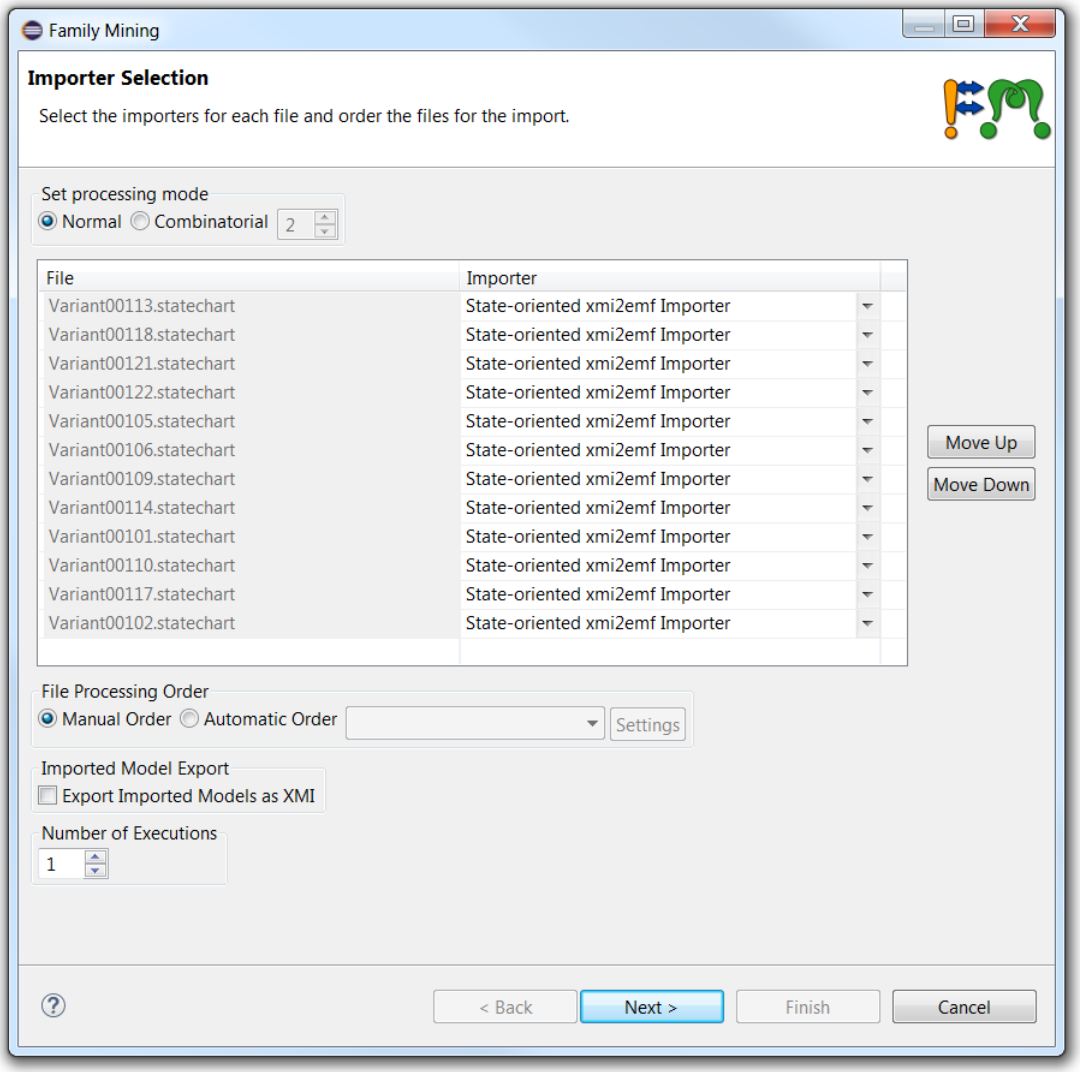


Figure 6.7.: Importer selection for the selected set of model variants.

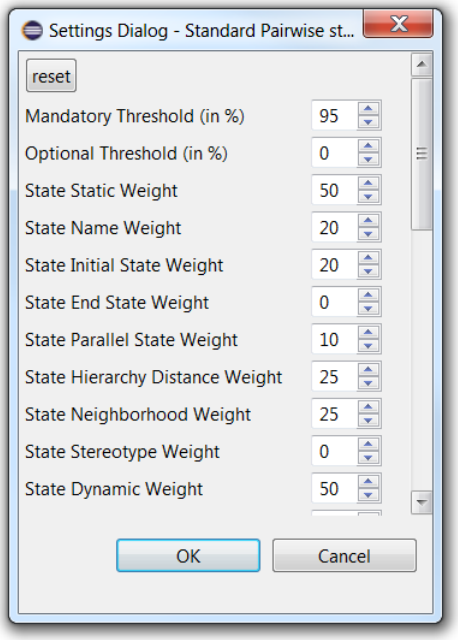


Figure 6.8.: Dialog to adjust weights and thresholds of the selected metric.

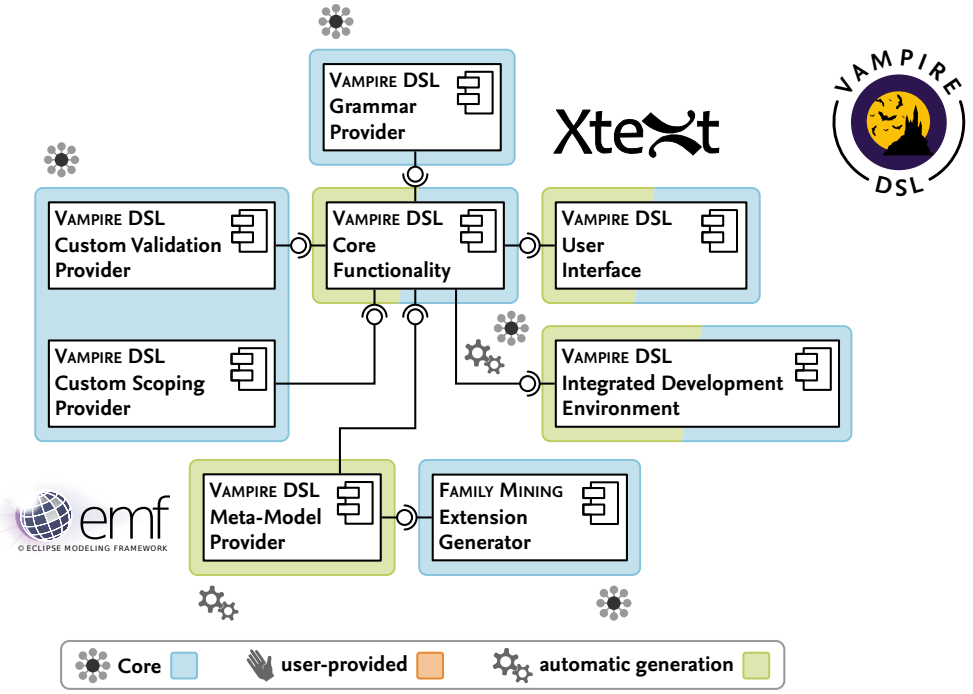


Figure 6.9.: Components of the Vampire DSL and its tooling.

6.3.1. General Realization of the Vampire DSL

In Figure 6.9, we show the component diagram for our Vampire DSL. The different components refer to plugs-ins providing functionality for the Vampire DSL and the generation of Family Mining plug-ins based on interpreted DSL descriptions. We highlight core parts in *blue* with a *core* pictogram, user-provided parts in *orange* with a *hand* symbol and automatically generated parts in *green* with *gears*. The core components comprise the Vampire DSL *Grammar Provider*, the Vampire DSL *Custom Validation Provider*, the Vampire DSL *Custom Scoping Provider* and the Family Mining *Extension Generator*. Based on the Vampire DSL grammar, it is possible to generate the Vampire DSL *Meta-Model Provider*, the Vampire DSL *Core Functionality*, the Vampire DSL *User Interface* and the Vampire DSL *Integrated Development Environment*. The last three components are partially contained in the core as the general plug-in structures exists and only their functionalities are generated.

The following list gives a detailed overview of all components together with their dependencies:

- Vampire DSL *Grammar Provider:*
  - *Description:* This component provides the grammar that is needed to derive the infrastructure for the Vampire DSL (i.e., parsing facilities and corresponding IDE support).
- Vampire DSL *Core Functionality:*
  - *Description:* This component provides the general parsing facilities for the Vampire DSL and all core functionality needed for its processing (e.g., in the generated IDE).
  - *Requires:*
    1. Vampire DSL *Grammar Provider:* Required to derive the complete facilities from the corresponding grammar.

2. Vampire DSL *Meta-Model Provider:* Required to access the model-based foundation of the Vampire DSL.
3. Vampire DSL *Custom Validation Provider:* Required to execute custom validations of Vampire DSL descriptions.
4. Vampire DSL *Custom Scoping Provider:* Required to correctly resolve references when parsing Vampire DSL descriptions.

- Vampire DSL *User Interface:*
  - *Description:* This component provides the user interface implementation of the Vampire DSL (e.g., the integration in the Eclipse wizard to create corresponding new files).
  - *Requires:*
    1. Vampire DSL *Core Functionality:* Required to access the core functionality of the Vampire DSL to correctly display parsed files in the editor.
- Vampire DSL *Integrated Development Environment:*
  - *Description:* This component provides the IDE for the Vampire DSL (e.g., corresponding editor support with highlighting).
  - *Requires:*
    1. Vampire DSL *Core Functionality:* Required to access the core functionality of the Vampire DSL to realize the IDE support.
- Vampire DSL *Custom Validation Provider:*
  - *Description:* This component provides additional validation facilities that we implemented to show custom error messages for illegal Vampire DSL descriptions.
- Vampire DSL *Custom Scoping Provider:*
  - *Description:* This component provides additional scoping facilities needed to correctly resolve references when parsing Vampire DSL descriptions.
- Vampire DSL *Meta-Model Provider:*
  - *Description:* This component is generated based on the Vampire DSL grammar and builds the model-based foundation of the Vampire DSL.
- Family Mining *Extension Generator:*
  - *Description:* This component interprets Vampire DSL descriptions and automatically generates corresponding Family Mining plugs-ins with meta-models, metrics and GUI integrations in the Family Mining framework.
  - *Requires:*
    1. Vampire DSL *Meta-Model Provider:* Required to access and process the parsed model-based Vampire DSL descriptions.

In Listing 6.1, we show an excerpt from the Xtext grammar for the Vampire DSL to give a general idea of the used grammar rules. The excerpt shows the concrete grammar rule for `Entities`, which can be `abstract` (i.e., an abstract meta-model class for this entity will be generated), `parameterizable` (i.e., the generated meta-model class inherits from the `ParameterizedElement` class in the base meta-model to allow generic parameterization) and `variable` (i.e., the class inherits from the base meta-model class `VariableElement` to allow annotation of variability). Furthermore, the specified `Entity` can inherit from other entities by extending them. The optional `EntityBody` allows to specify different properties in form of `Attributes`, `References` and `MapReferences`. For concrete examples of Vampire DSL descriptions in this grammar, we refer to Listing A.1, Listing C.1 and Listing D.1 in the appendix.

```
1  grammar de.tu_bs.cs.isf.familymining.tooling.vampire.VampireDsl with
2    org.eclipse.xtext.common.Terminals
3
4  generate vampireDSL
5    "http://www.tu_braunschweig.de/isf/familymining/vampire_dsl"
6
7  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
8
9  // ...
10
11 Entity:
12   (abstract ?= 'abstract')?
13   (parameterizable ?= 'parameterizable')?
14   (variable ?= 'variable')?
15   'Entity' name = ID
16   (
17     'extends' extendedElements += [AbstractEntity]
18     (',' extendedElements += [AbstractEntity])*
19   )?
20   ('{' entityBody = EntityBody '}')?
21 ;
22
23 EntityBody:
24   {EntityBody} (properties += Property)*
25 ;
26
27 Property:
28   Attribute | Reference | MapReference
29 ;
30
31 // ...
```

Listing 6.1: Excerpt from the Xtext grammar for the Vampire DSL.

Based on this grammar, we are able to derive parsing facilities with GUI and IDE support using the Xtext framework. The generated facilities allow users to specify meta-models and concrete metrics by defining Vampire DSL descriptions (cf. Chapter 4) to adapt our Family Mining algorithms for new languages. This way, we largely reduce the effort for adapting the fine-grained variability mining discussed in this thesis and, thus, lower the barrier of applying it for new languages.

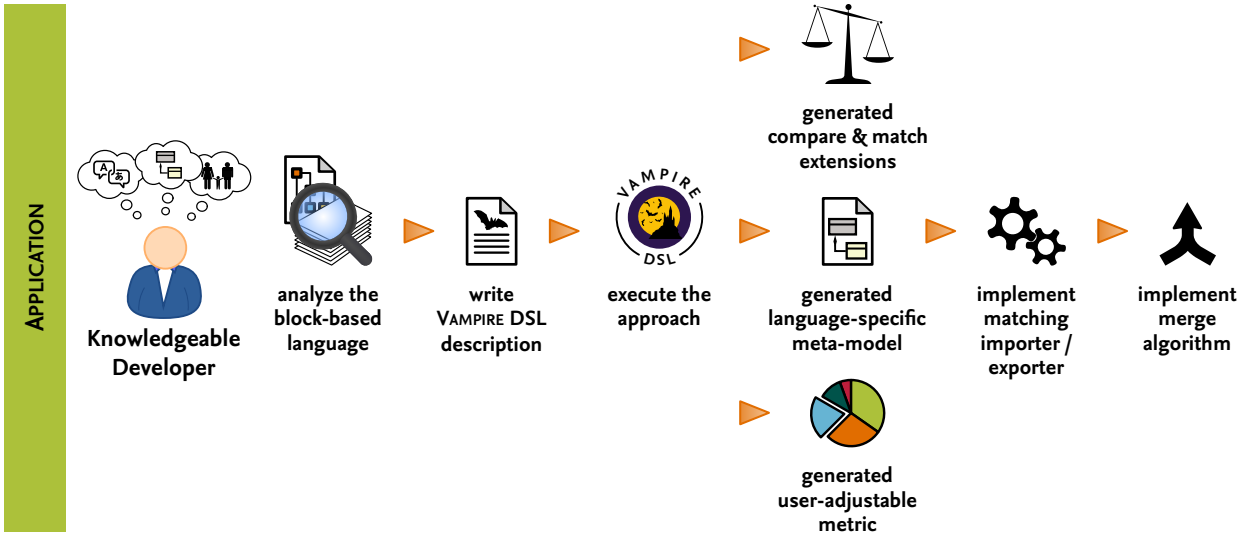


Figure 6.10.: Workflow to apply the Vampire DSL to adapt Family Mining for new languages.

6.3.2. Adapting and Applying Variability Mining using the Vampire DSL

In Figure 6.10, we show the workflow on how to apply the Vampire DSL to adapt our generic Family Mining algorithms for new languages. Similar to the adaptation of the Family Mining approach (cf. Section 6.2.2), we assume that a *knowledgeable developer* is available. This user should have at least basic knowledge of meta-modeling, the used modeling language and the product family that should be analyzed with the Family Mining approach. This developer is most likely a senior developer with experience and a general overview of project goals in the analyzed model variants. Similar to the manual adaptation workflow for the Family Mining approach, the developer has to analyze the block-based language to gather knowledge (cf. Chapter 4) for executing the subsequent steps. Based on this knowledge, the user can specify a language-specific meta-model and corresponding metrics (cf. Section 4.2.4, Section 4.3.3 and Section 4.6.1). Using the Vampire DSL generator, the developer is able to derive large parts of the needed extensions for the generic Family Mining algorithms and only a corresponding importer, exporter and merge algorithm have to be manually implemented for a complete adaptation.

6.4. Migrating to a Software Product Line using the Matador SPL Approach

To provide tooling for an automated migration of existing model variants to a delta-oriented SPL, we implemented the Matador SPL approach as a direct extension of the Family Mining framework. In addition to the automatic migration, this extension also provides tooling to refactor the generated SPL artifacts. Thus, it allows developers to directly migrate their model variants to an SPL based on the results of a fine-grained variability analysis using the Family Mining algorithms (cf. Chapter 4) with the possibility to apply manual adjustments of the generated artifacts. The Matador SPL approach is realized using Java and relies on the Eclipse RCP facilities as it extends the Family Mining framework through its extension points (cf. Section 6.2). For the migration, the Matador SPL approach relies on the EMF and DeltaEcore for the concrete SPL artifacts.

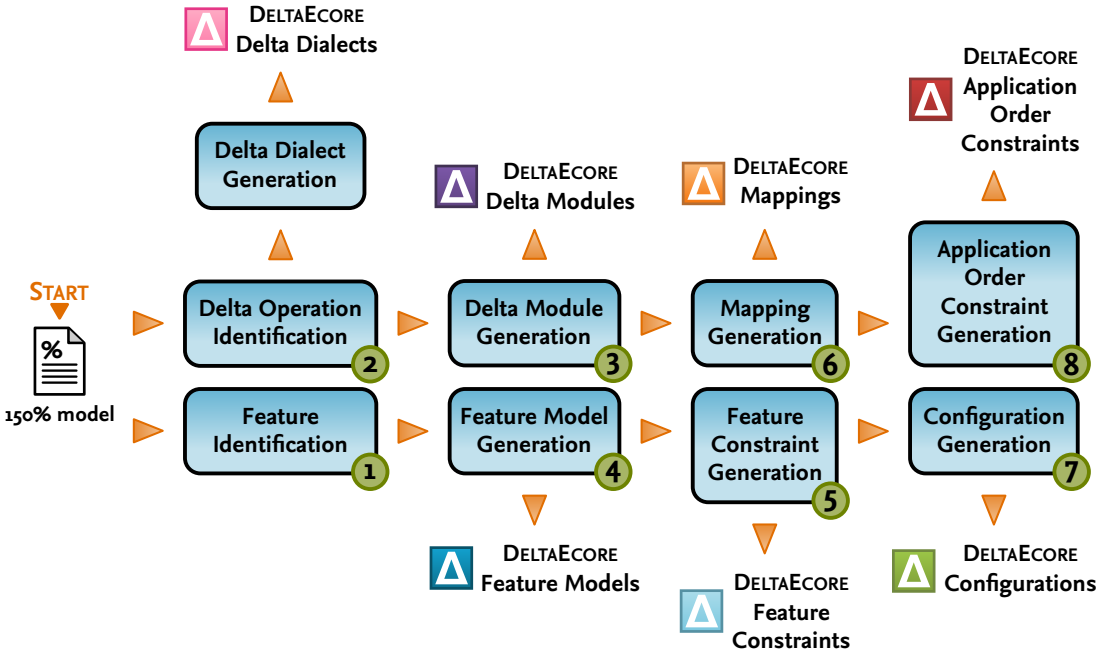


Figure 6.11.: Steps of the processing pipeline in the Matador SPL approach.

6.4.1. General Realization of the Matador SPL Approach

In Figure 6.11, we show the processing pipeline of the Matador SPL approach. The *Delta Dialect Generation* only requires the information on required delta operations provided by the *Delta Operation Identification Algorithms* to generate a corresponding DeltaEcore delta dialect for the current language-specific meta-model (upper left part of the figure). In contrast, the *Delta SPL Exporter* has to ensure that the required information is available to generate the different artifacts and, thus, only executes them in such cases. For the actual generation, the corresponding steps are executed in the annotated order in Figure 6.11 (the green numbered circles). In general, it is also possible to execute only parts of the Matador SPL approach (e.g., if certain information, such as details on the features, is not available) as the *Delta SPL Exporter* always ensures that all required information for a step is available and otherwise skips its execution.

In Figure 6.12, we show the component diagram for the Matador SPL approach. The different components refer to plugs-ins providing functionality for the realization of the Matador SPL approach. We highlight core parts in *blue* with a *core* pictogram, user-provided parts in *orange* with a *hand* symbol and automatically generated parts in *green* with *gears*. The larger part of the Matador SPL components are contained in the core implementation. Only small parts have to be generated or manually provided. We highlight that the required *Base Meta-Model Provider* is contained in the Family Mining framework (cf. Section 6.2) and the *Language-Specific Meta-Model Provider* can be generated using the Vampire DSL (cf. Section 6.3). In addition, the *Language-Specific Delta Dialect Provider* can be generated using the *Delta Dialect Exporter* from the Matador SPL approach (cf. Section 5.2). The only required user-provided parts for the Matador SPL approach contain additional details for the *Feature Identification Algorithm* in form of *Family-Specific Feature Identification Extensions* providing details on how to identify features in the processed 150% model (cf. Section 5.4). In case of the *Refactoring Operations Provider* for the refactoring tooling of the Matador SPL approach, the

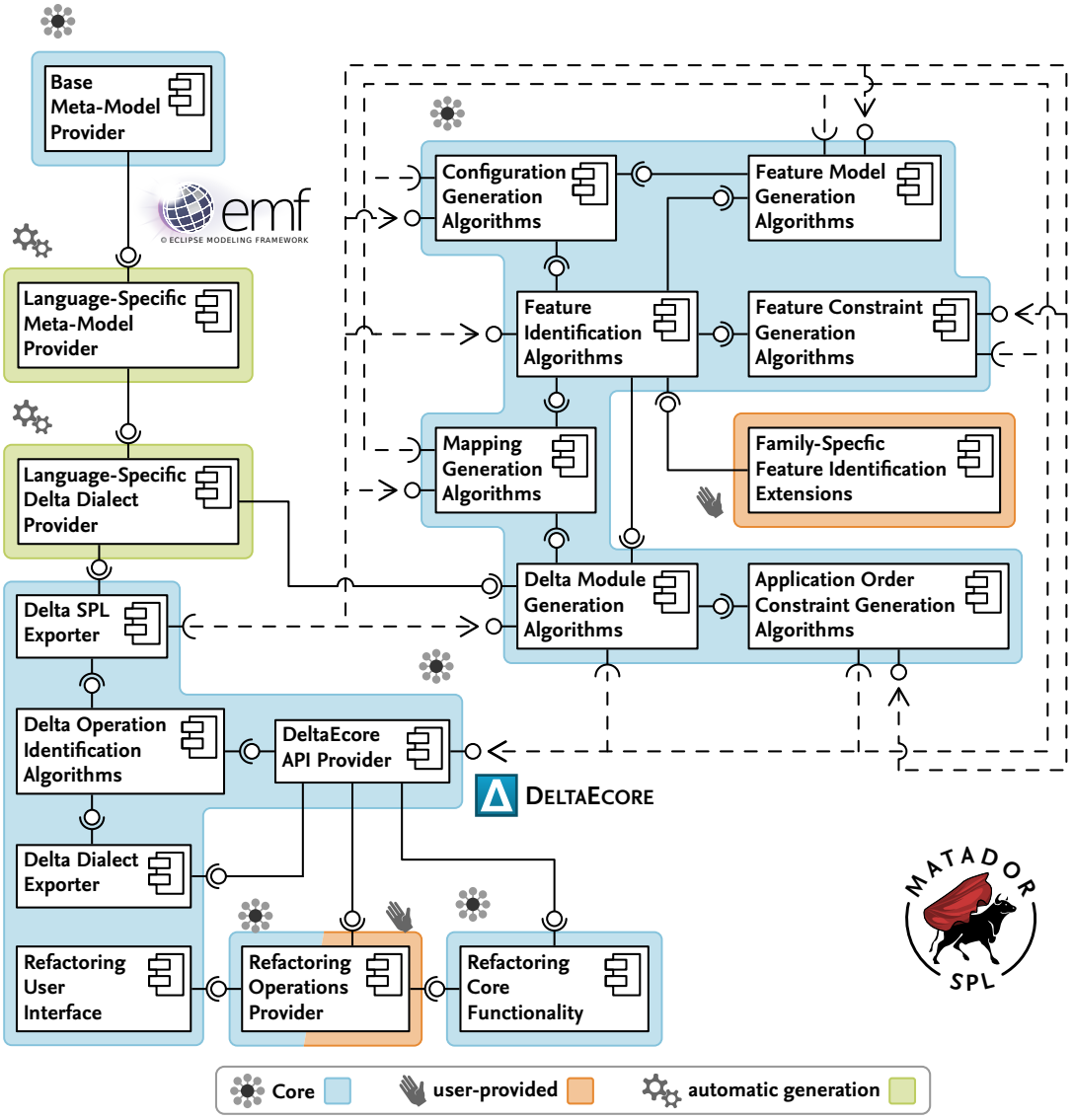


Figure 6.12.: Components of the Matador SPL approach.

user-provided parts are optional as they allow integration of user-specified operations in addition to the existing core operations (cf. Table 5.3). All components executing the generation of SPL artifacts for the *Delta SPL Exporter* (i.e., the components in the upper right part of Figure 6.12) are triggered via corresponding extension points to allow easy replacement with custom algorithms (e.g., a different approach to identify features). Both the *Delta Dialect Exporter* and the *Delta SPL Exporter* are direct extensions to the exporter extension point of the Family Mining framework (cf. Section 6.2).

The following list gives a detailed overview of all components together with their dependencies to other components where applicable:

- *Base Meta-Model Provider:*
  - *Description:* This component provides the base meta-model, which can be used as the core of meta-models for our generic Family Mining. Thus, it is contained in the core of the Family Mining framework and we leave a detailed discussion to Section 6.2.

- *Language-Specific Meta-Model Provider:*
  - *Description:* This component provides a language-specific meta-model, which can be generated using the Vampire DSL of the Family Mining framework and we leave a detailed discussion to Section 6.2.
  - *Requires:*
    1. *Base Meta-Model Provider:* This requirement is optional as meta-models only rely on the base meta-model when using corresponding parts.
- *Language-Specific Delta Dialect Provider:*
  - *Description:* This component provides a language-specific delta dialect allowing modification of model instances using the language-specific meta-model by applying corresponding delta operations from the delta dialect.
  - *Requires:*
    1. *Language-Specific Meta-Model Provider:* Required as it provides the language-specific meta-model whose model instances should be modified by the delta dialect.
- *Delta Operation Identification Algorithm:*
  - *Description:* This component provides the algorithm to identify required delta operations from a 150% model showing variability relations between variants (cf. Section 5.1).
  - *Requires:*
    1. DeltaEcore *API Provider:* Required as it provides access to the DeltaEcore API to use corresponding structures for the representation of identified operations.
- *Delta Dialect Exporter:*
  - *Description:* This component extends the Family Mining exporter extension point (cf. Section 6.2) and derives delta dialects from identified delta operations (cf. Section 5.2).
  - *Requires:*
    1. *Delta Operation Identification Algorithms:* Required as it provides the delta operations identified for a 150% model storing the variability for a set of model variants.
    2. DeltaEcore *API Provider:* Required as it provides access to the DeltaEcore API to use corresponding structures for generating DeltaEcore dialects.
- *Delta SPL Exporter:*
  - *Description:* This component is realized as an extension to the Family Mining exporter extension point (cf. Section 6.2) and allows migration of identified variability relations for a set of model variants to a delta-oriented SPL (cf. Section 5.3 and Section 5.4). This exporter itself defines further extension points, which allow integration of plug-ins executing the different steps of the SPL migration (e.g., feature models or delta modules).
  - *Requires:*
    1. *Delta Operation Identification Algorithms:* Required as it provides the delta operations identified for a 150% model storing the variability for a set of model variants.

2. *Feature Identification Algorithms:* Required to identify features for the generated SPL realization in the hierarchy of the 150% model.
3. *Delta Module Generation Algorithms:* Required to encode the identified variability from the 150% model in the generated SPL.
4. *Mapping Generation Algorithms:* Required to generate a mapping between features and corresponding realizations in form of delta modules.
5. *Configuration Generation Algorithms:* Required to generate configuration for the analyzed model variants in the 150% model.
6. *Feature Model Generation Algorithms:* Required to generate a feature model containing the identified features.
7. *Feature Constraint Generation Algorithms:* Required to generate recommendations for constraints between identified features.
8. *Application Order Constraint Generation Algorithms:* Required to generate application orders for delta modules.

- *Feature Identification Algorithms:*
  - *Description:* This component provides the algorithms to identify features in the hierarchy of 150% models and defines an extension point for family-specific identification details.
  - *Requires:*
    1. *Family-Specific Feature Identification Extensions:* Required as it provides information to identify relevant features.
- *Family-Specific Feature Identification Extensions:*
  - *Description:* This component provides user-specified information on how to identify relevant features for the current model family (e.g., which model elements to consider).
- *Delta Module Generation Algorithms:*
  - *Description:* This component provides the generation algorithms for delta modules and either generates delta modules for complete variants or identified features.
  - *Requires:*
    1. DeltaEcore *API Provider:* Required as it provides access to the necessary data structures of DeltaEcore to generate delta modules.
    2. *Feature Identification Algorithms:* This requirement is optional for the generation of delta modules per feature. Without this additional information the algorithm generates delta modules per variant.
    3. *Language-Specific Delta Dialect Provider:* Required as it provides the delta dialect that has to be used for the current meta-model to encode the variability in delta modules.
- *Mapping Generation Algorithms:*
  - *Description:* This component provides the generation algorithms for a mapping between identified features and their corresponding delta modules.

- *Requires:*
  1. DeltaEcore *API Provider:* Required as it provides access to the necessary data structures of DeltaEcore to generate mappings between features and delta modules.
  2. *Delta Module Generation Algorithms:* Required as it provides the delta modules that are mapped to features.
  3. *Feature Identification Algorithms:* Required as it provides the features, which are linked with corresponding implementations in form of the delta modules.

- *Feature Model Generation Algorithms:*
  - *Description:* This component provides the generation algorithms to derive features models from the identified features.
  - *Requires:*
    1. DeltaEcore *API Provider:* Required as it provides access to the necessary data structures of DeltaEcore to generate feature models.
    2. *Feature Identification Algorithms:* Required as it provides the features that have to be contained in the feature model hierarchy.

- *Configuration Generation Algorithms:*
  - *Description:* This component provides generation algorithms to create configurations for the variants that served as input to the Family Mining.
  - *Requires:*
    1. DeltaEcore *API Provider:* Required as it provides access to the necessary data structures of DeltaEcore to generate configurations.
    2. *Feature Identification Algorithms:* Required as it provides the features that are contained in the configurations of the variants.
    3. *Feature Model Generation Algorithms:* Required as it provides dependencies (e.g., parent-child relations) between features.

- *Feature Constraint Generation Algorithms:*
  - *Description:* This component provides generation algorithms to derive recommendations for constraints between the identified features.
  - *Requires:*
    1. DeltaEcore *API Provider:* Required as it provides access to the necessary data structures of DeltaEcore to generate constraints.
    2. *Feature Identification Algorithms:* Required as it provides the features whose constraints should be identified.

- *Application Order Constraint Generation Algorithms:*
  - *Description:* This component provides generation algorithms to derive the application order for the created delta modules.

- *Requires:*
  1. DeltaEcore *API Provider:* Required as it provides access to the necessary data structures of DeltaEcore to generate application order constraints.
  2. *Delta Module Generation Algorithms:* Required as it provides the delta modules whose application order has to be determined.

- *Refactoring Core Functionality:*
  - *Description:* This component provides basic capabilities, such as importing files and triggering actions, for the Matador SPL refactoring tooling.
  - *Requires:*
    1. DeltaEcore *API Provider:* Required as it provides access to the DeltaEcore API to use corresponding structures importing realization artifacts (e.g., feature models and delta modules) of SPLs.

- *Refactoring Operations Provider:*
  - *Description:* This component provides the core refactoring operations of the Matador SPL approach (cf. Section 5.5) via a corresponding extension point and, thus, can be extended with user-specified operations.
  - *Requires:*
    1. DeltaEcore *API Provider:* Required as it provides access to the DeltaEcore API to allow refactoring of imported realization artifacts (e.g., feature models and delta modules) of SPLs.
    2. *Refactoring Core Functionality:* Required as it provides access to the core functionality of the Matador SPL refactoring tooling (e.g., import of files).

- *Refactoring User Interface:*
  - *Description:* This component provides the GUI of the Matador SPL refactoring tool.
  - *Requires:*
    1. *Refactoring Operations Provider:* Required as it provides the refactoring operations that should be selectable from the GUI.

Using the described architecture it is possible to migrate existing model variants based on a 150% model created by the Family Mining algorithms to a delta-oriented SPL and refactor it to user-specific requirements. The approach requires almost no user-specified input and executes all steps fully automatically. Furthermore, the only user input is optional as it comprises details on features contained in the analyzed variants, which can be provided using an easily understandable interface to specify relevant model elements and their names. And even if this information is not available, the Matador SPL approach is still applicable as it would be still possible to generate delta modules per analyzed variant. Thus, the realization of the Matador SPL approach is easily applicable for different settings without additional complex knowledge.

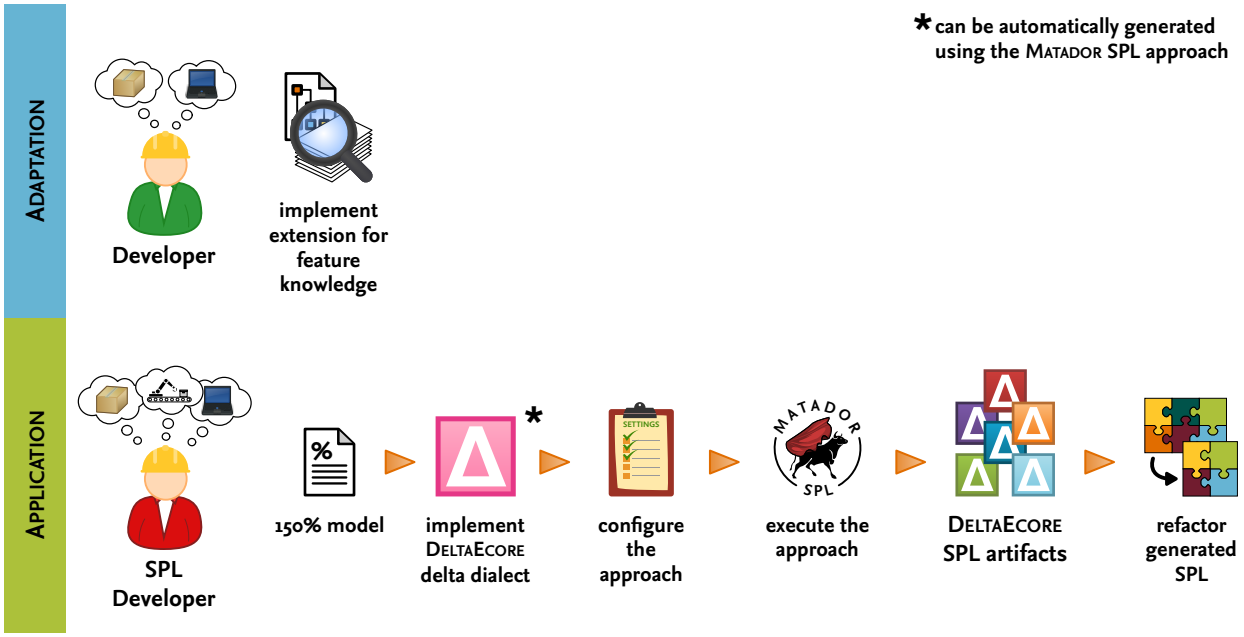


Figure 6.13.: Workflows to adapt and apply Matador SPL approach for new settings.

6.4.2. Adapting and Applying the Matador SPL Approach

In Figure 6.13, we show the workflow on how to adapt the Matador SPL approach for different settings and how to apply it to derive corresponding delta-oriented SPLs for analyzed model variants.

For the adaptation of the Matador SPL approach for a new project (e.g., when migrating a new set of product variants to an SPL), we assume that a *developer* with at least basic knowledge of the developed products is available. Using their knowledge of naming conventions and the basic structure of the developed products (i.e., the structures encapsulating specific model features), these developers implement extensions for the Matador SPL approach allowing identification of features. However, this step is not mandatory as the migration to an initial SPL realization is also able without analyzing the products' features (i.e., delta modules for complete variants are generated).

For the application of the Matador SPL approach, we assume that an *SPL developer* is available. This developer should at least have superficial knowledge of the products that are migrated to an SPL, know the used modeling language, understand the concepts of SPLs and know the company's goals for the created SPL. Starting with a 150% model (e.g., created using our Family Mining approach), the developer implements a DeltaEcore delta dialect (and generates the corresponding DeltaEcore delta language – cf. Section 5.2) or, first, applies the Matador SPL approach to automatically derive it (thus, we marked it with an asterisk in Figure 6.13). This dialect allows to express the variability between the variants in the 150% model. For this automatic generation, the developer uses the same steps as for the remaining SPL migration, but selects the delta dialect generation instead. Thus, we do not further discuss this workflow. Next, the developer configures the approach (i.e., whether to analyze the product features) and executes the Matador SPL approach. The generated DeltaEcore artifacts can afterwards be refactored based on the developer's knowledge to create an SPL conforming to the company's requirements. Also it is possible to rename operations in the generated delta dialect as long as their signature remains the same (cf. Section 5.2).

6.5. Chapter Summary

Using the described Family Mining framework allows to easily apply detailed analysis of variability relations between model variants in different modeling languages and subsequent migration of the variants to an SPL. By creating descriptions for the necessary language-specific adaptations using our Vampire DSL, users can easily generate the required meta-models together with metrics and extensions for the generic Family Mining algorithms. Thus, the manual effort for adapting our approaches is limited to implementing a small number of components using the clearly defined interfaces of the Family Mining framework.

Overall, the structure of the provided Family Mining framework components and the underlying generic algorithms support easy adaptation of the approaches discussed in Part II of this thesis for new languages. Furthermore, we demonstrated that the general design of the Family Mining framework can also be used as an easily extensible basis and support for further variability mining algorithms in other contexts [SWC+17, WWS+17c].

7 Evaluation

This chapter's contents are largely based on work published in [WBC+18, WSS+16, WSS16, WRS+17, SWS+17].

The evaluation demonstrates the feasibility of the techniques described within this thesis. First, we demonstrate that it is feasible to analyze coarse-grained variability relations (i.e., clusters and outliers) for large sets of model variants with potentially unknown relations by using our CoreVid approach (cf. Chapter 3). Furthermore, we demonstrate that fine-grained variability relations can be identified and analyzed using the 150% models generated by our Family Mining approach (cf. Chapter 4). Based on these results, we demonstrate that the identified variability relations in form of corresponding 150% models can be used to automatically migrate the analyzed variants to an SPL using our Matador SPL approach (cf. Chapter 5). During the evaluation, we analyze different requirements that are relevant with respect to a detailed variability analysis of existing model variants using our proposed solutions:

- *Correctness and Precision:* The correctness and precision of generated results is key for useful variability information identified by any reverse-engineering algorithm. Otherwise, incorrect variability relations inevitably compromise subsequent steps and, thus, undermine the users' trust in the applied approach. For example, migrating to an SPL realization based on incorrect variability relations results in an erroneous realization of the variants' variability in the corresponding SPL artifacts. As a result, users would not use the proposed techniques, because they cannot rely on them. Thus, analyzing the results of our algorithms with respect to their correctness is an important factor for our evaluation.

- *Runtime and Scalability:* Further key aspects for reverse-engineering algorithms are their runtime and scalability. Sensible runtimes for analyzing the complete fine-grained variability relations in complex systems are essential for a productive use of such algorithms. Another important factor is scalability of the used approaches as they have to be reliably applicable to realistic industrial-scale models in order to be useful to domain experts. If either the runtimes are not acceptable or the approach does not scale, it most likely will not be accepted by developers as it cannot be applied in their daily work. Thus, analyzing the runtime and scalability of our algorithms is the second important factor for our evaluation.

- *Adaptability:* One of our main goals in this thesis is the adaptability of our proposed techniques for different settings. Without the corresponding support, developers would have to manually reimplement our techniques for their specific requirements. Thus, analyzing the adaptability of our algorithms for different scenarios and new languages with low effort is another key factor for the evaluation of our proposed solutions.

- *Usefulness of the Results:* Another important factor is the usefulness of the results generated by our approaches. Without analyzing this aspect, the created results might be correct and

scalable for large scenarios in different languages, but might not actually solve the targeted problems. Thus, considering the usefulness of our results is necessary to understand whether they are applicable for the intended use cases.

Chapter Outline In the following sections, we explain the details of our evaluation:

- *Section 7.1:* This section gives details about the research questions answered during the evaluation of our proposed techniques.
- *Section 7.2:* This section gives details about the case study subjects used to evaluate different aspects of the proposed techniques.
- *Section 7.3:* This section gives details about the used methodology to collect the data needed to answer the research questions.
- *Section 7.4:* This section presents the results of our evaluation and corresponding discussions with respect to our research questions.
- *Section 7.5:* This section gives details about expert interviews that we executed with an industry partner. Our goal was to understand to what extent the domain experts' requirements regarding variability mining are met by our approach.
- *Section 7.6:* This section discusses threats to validity that apply to our approach and their evaluation, although we designed, implemented and evaluated our techniques with great care.

7.1. Research Questions

We define our research questions based on the four requirements that we identified to be relevant for our evaluation (cf. the introduction of this chapter). For the concrete evaluation, we investigate the following research questions to demonstrate the actual feasibility of our proposed techniques:

RQ1 Correctness and Precision
Are the proposed techniques capable of providing correct and precise results?

RQ1.1a CoreVID – *Outlier Detection*
Is the clustering technique capable of eliminating outliers in input models before executing the Family Mining approach?

RQ1.1b CoreVID – *Cluster Detection*
Is the clustering technique capable of identifying sensible clusters of related input models for the Family Mining approach?

RQ1.2 Family Mining – *Level of Correctness of Results*
What level of precision and recall can we achieve with our Family Mining approach?

RQ1.3a Matador SPL – *Correctness of Delta Languages and Modules*
Are the generated delta languages and delta modules capable of correctly transforming an SPL core variant into the corresponding target variants?

RQ1.3b Matador SPL – *Feature Dissection*
Is the Matador SPL approach capable of correctly dissecting the generated SPL into various independent features with associated realizations?

RQ2 Runtime and Scalability
Are the proposed techniques scalable for realistic scenarios with sensible runtimes?

RQ2.1 Corevid – *Runtime and Scalability*
Is the Corevid approach scalable for realistic scenarios with sensible runtimes?

RQ2.2 Family Mining – *Runtime and Scalability*
Is the Family Mining approach scalable for realistic scenarios with sensible runtimes?

RQ2.3 Matador SPL – *Runtime and Scalability*
Is the Matador SPL approach scalable for realistic scenarios with sensible runtimes?

RQ3 Adaptability
Is it feasible to successfully adapt our proposed techniques for different block-based languages with varying concepts?

RQ3.1 Corevid – *General Adaptability*
Is it feasible to successfully adapt identification of coarse-grained variability relations for different block-based languages using our Corevid approach?

RQ3.2a Family Mining – *General Adaptability*
Is it feasible to successfully adapt identification of fine-grained variability relations for different block-based languages using our Family Mining approach?

RQ3.2b Family Mining – *Reduction of Implementation Effort*
Does our generic framework reduce the implementation effort of adapting our Family Mining for new languages compared to writing a custom Family Mining solution?

RQ3.3 Matador SPL – *Language Independence*
Is the proposed Matador SPL approach capable of generating delta languages and delta modules for different modeling languages used to realize cloned product variants?

RQ4 Usefulness of Results
Are the results generated by our proposed techniques useful in the described scenarios?

RQ4.1 Corevid – *Improvement of Results*
Are the fine-grained variability relations improved by our Corevid approach when applying the Family Mining approach only to identified clusters and neglecting outliers?

RQ4.2 Family Mining – *Usefulness of Results*
Is our Family Mining capable of providing variability information supporting domain experts in an industrial environment during their daily work?

RQ4.3 Matador SPL – *Usefulness of Results*
Is our Matador SPL approach capable of generating SPL artifacts that can be used as a solid basis for migrating existing model variants to an SPL?

7.2. Subjects

For the evaluation of our approaches, we used three case studies from the automotive domain. Two of these case studies (cf. Section 7.2.1 and Section 7.2.2) are publicly available, while the other one (cf. Section 7.2.3) was provided by one of our industry partners.

7.2.1. Body Comfort System Case Study

As our first subject for the evaluation of our approaches, we used the publicly available *Body Comfort System (BCS)* case study, which is implemented in form of IBM Rational Rhapsody statecharts [LLL+13]. This case study is based on a real-world industrial implementation of a *BCS* in a car that never went into production. In their work on pairwise feature interaction testing, Oster et al. [OZL+11] decomposed the original *BCS* into a corresponding SPL realization with variable parts. Furthermore, the authors defined 17 representative product variants `P1` – `P17` comprising combinations of this variability that were later extended with their common core variant `P0` by Lity et al. [LLL+13]. The case study comprises 27 features (e.g., an Alarm System or a Finger Protection for the windows) with five requires dependencies between specific features (e.g., that the status LED for the heatable exterior mirrors requires the actual heatable exterior mirrors) and one excludes dependency (i.e., that the manual power window cannot be used together with the power window control of the remote control key). The SPL implementation of the *BCS* allows generation of 11,616 valid product variants when considering all possible feature configurations. The generated statechart variants contain up to 283 model elements (i.e., regions, states and transitions).

To provide a solid basis for evaluations within different research areas (e.g., the case study also comprises corresponding test cases), the *BCS* case study was realized following SPL best practices. Thus, it can serve as a reliable ground truth for the evaluation of different aspects regarding our proposed variability mining algorithms and, most importantly, a corresponding migration to an SPL. We have used parts from the *BCS* throughout the thesis as running examples (cf. Section 2.2) to illustrate our algorithms and allow better understanding of their general ideas. However, all ideas and algorithms were developed independently from the concrete implementation of the *BCS*, and all references prior to this chapter were used for illustrative purposes only. Thus, the algorithms described in this thesis are not biased by an overfitting for the concrete scenarios of the *BCS*.

BCS1 – Selected Subjects for Evaluating the Corevid Approach To evaluate the *outlier and cluster detection capabilities* of our Corevid approach, we define five scenarios based on *BCS* variants with varying feature configurations to simulate different scenarios. In Table 7.1, we show a summary of the selected scenarios. The first three scenarios (i.e., `OD1` – `OD3`) aim at evaluating the outlier detection capabilities of the Corevid approach and were designed to detect a small number of variants within a set of otherwise highly related variants. The selected outlier variants (i.e., *Cluster 2*) differ from the remaining variants (i.e., *Cluster 1*) as parts of their selected features are different. Furthermore, to evaluate the cluster detection capabilities of the Corevid approach without additional outliers present, we define two corresponding scenarios (i.e., `CD1` and `CD2`). In these scenarios, *Cluster 1* and *Cluster 2* refer to clearly delimitable clusters of highly related variants. All selected scenarios comprise valid feature selections from the *BCS* case study to allow generation of the corresponding variants for the evaluation.

In Table 7.1, we show the number of shared features (i.e., contained in both clusters), the number of mutually exclusive features (i.e., contained only in one of the clusters) and the number of alternating features (i.e., features that cannot clearly be assigned to variants from a particular cluster). Overall, the selected scenarios allow us to evaluate our Corevid approach using variants with different degrees of similarity (i.e., number of shared or mutually exclusive features). In addition, incorporating additional alternating features allows us to evaluate the resistance of the technique

	Statechart Variants			Shared	Mutually Exclusive	Alternating
	Cluster 1	*Cluster 2**	∑	**Features**	**Features**	**Features**
OD1	8	2	10	7	14	6
OD2	8	2	10	10	12	5
OD3	8	2	10	6	0	21
CD1	8	8	16	17	2	8
CD2	12	12	24	12	7	8

*containing the *outliers* for the OD scenarios and the *second cluster* for the CD scenarios

Table 7.1.: High-level overview of five scenarios from the *Body Comfort System (BCS)* used to evaluate the cluster detection (CD1 & CD2) and outlier detection (OD1 – OD3) capabilities of the Corevid approach.

against "noise" that might negatively influence the Corevid approach as variants might be assigned to unexpected clusters. Using the selected scenarios provides us with the necessary ground truth (i.e., clearly labeled outlier variants or which cluster the variants belong to) for the evaluation. As a consequence, we can evaluate whether the detection is capable of generating results conforming to this ground truth and, thus, meeting the expectations of experts well familiar with the *BCS* implementation. In addition, the selected scenarios allow us to evaluate the *runtimes* and the *scalability* of our Corevid approach (i.e., the coarse-grained variability detection) as well as our Family Mining approach (i.e., the fine-grained variability detection) for these scenarios. In the remainder of the evaluation, we refer to these selected model variants as the *BCS1* subjects.

BCS2 – Selected Subjects for Evaluating the Family Mining Approach To evaluate the *correctness* of our Family Mining for variability mining of statecharts, we selected 27 pairwise comparisons from the 18 product variants (i.e., P0 – P17). From the executed 27 pairwise comparisons, the first 17 compare the product variants P1 – P17 with product variant P0. Comparing with this common basis of all other product variants allows us to evaluate whether the variability in relation to the core is identified correctly. Here, all additional features extending the core have to be identified as optional parts with respect to the *BCS SPL*. All other combinations compare product variants containing different variations of the same feature. For instance, P1 and P11 contain alternative variations of the Central Locking System (CLS) feature. By applying our adapted family mining algorithms to these cases, we evaluate whether these alternatives are identified correctly. Overall, the *BCS* case study provides us with a ground truth of the variability that our Family Mining approach should detect and allows easy detection of potential errors in the variability mining results. In addition, the selected 27 comparisons allow us to evaluate the *runtimes* and the *scalability* of our Family Mining approach for these scenarios. In the remainder of the evaluation, we refer to these selected model variants as the *BCS2* subjects.

BCS3 – Selected Subjects for Evaluating the Matador SPL Approach To evaluate our Matador SPL approach, we used the *BCS1* subjects and *BCS2* subjects and summarized them as the *BCS3* subjects. This way, we are able to evaluate whether our Matador SPL approach is capable of *correctly* generating the corresponding delta languages comprising all necessary delta operations. In addition, we can analyze whether the Matador SPL approach *correctly* encodes the identified variability in

corresponding delta modules based on such delta languages and, thus, allows derivation of all input variants. Overall, combining the subjects from the *BCS1* and *BCS2* model variants allows us to check the SPL migration capabilities from scenarios with low complexity (i.e., the *BCS2* subjects) up to complex scenarios with multiple variants (i.e., the *BCS1* subjects). In addition, these scenarios allow us to evaluate the *runtimes* and the *scalability* of our Matador SPL approach.

7.2.2. Driver Assistance System Case Study

As our second subject for the evaluation of our approaches, we used the publicly available real world Matlab/Simulink models from the *SPES_XT*[1] project. These models implement the functionality of a *Driver Assistance System (DAS)* in a car. Based on the models, we identified and extracted a set of submodels that represent the functionality of clearly delimitable features in the *DAS*. In Table 7.2, we present these features together with details on their size (i.e., number of blocks), number of subsystems and hierarchy depth. In addition, we annotated two dependencies that we identified for the features FollowToStop (*'FS'*) and Distronic (*'DT'*) using the available project documentation. These features have to be used together with the CruiseControl ('CC') feature as it provides crucial base functionality for them (i.e., the possibility to maintain a driver-selected speed).

Observing the identified dependencies between the features, it is possible to artificially generate 19 *DAS* variants from these feature submodels. The largest of the generated variants comprises all feature submodels, while other variants might only contain a subset of them. These variants address a *model adaptation scenario* where developers adapt an existing variant to new requirements and, thus, are similar to clone-and-own scenarios. For example, a scenario where a variant comprising only a single feature submodel (e.g., the *'FS'* feature submodel) with a variant comprising two of the feature submodels (e.g., the *'FS'* and the *'DT'* feature submodels) could be an extension of the first variant.

DAS1 – Selected Subjects for Evaluating the Family Mining Approach In theory, it is possible to execute 171 pairwise comparisons[2] for these 19 model variants. However, as our main goal for this case study was to execute manual evaluation of the results to check for the *correctness* of the fine-grained variability information identified by our Family Mining, this would not be feasible. Thus, we limited our manual analysis to a subset of 51 randomly selected model comparisons (i.e., $\approx 30\%$ of all possible pairwise comparisons). Further goal is to evaluate the *runtimes* and the *scalability* of our Family Mining approach for these scenarios. In the remainder of the evaluation, we refer to these selected model variants as the *DAS1* subjects.

DAS2 – Selected Subjects for Evaluating the Matador SPL Approach In addition, we selected eight scenarios with pairwise model variants from the *DAS*, which have increasing differences between their comprised feature submodels to evaluate our Matador SPL approach. Starting with two model variants that are almost alike and only differ in one contained feature submodel, we increase the differences up to a case where the two model variants only share one feature submodel and differ in the other feature submodels. The rational behind this is to have variability scenarios with increasing complexity and, thus, potentially different scenarios regarding the delta operations required by the needed delta language and the variability expressed in delta modules encoding the corresponding variability. This way, we are able to evaluate the *correctness* of our Matador SPL approach in different settings (i.e., whether the required delta operations are contained in the generated delta

[1] `http://spes2020.informatik.tu-muenchen.de/spes_xt-home.html`

[2] $(19 \times 18)/2 = 171$, because the input order does not matter for pairwise comparisons

Feature Submodels	Abbreviation	Blocks	Subsystems	Hierarchy Depth
EmergencyBreak	*'EB'*	409	43	7
FollowToStop*	*'FS'*	699	77	11
SpeedLimiter	*'SL'*	497	57	10
CruiseControl	*'CC'*	671	74	11
Distronic*	*'DT'*	728	78	11

* = *these features require the* CruiseControl *feature 'CC'*

Table 7.2.: Feature submodels extracted from the *Driver Assistance System (DAS)* in the *SPES_XT* project.

language and the generated delta modules allow correct derivation of the input variants). In addition, we are able to evaluate corresponding *runtimes* and the *scalability* for generating delta languages and delta modules with our Matador SPL approach for these scenarios. In the remainder of the evaluation, we refer to these selected model variants as the *DAS2* subjects.

7.2.3. Industrial Case Study

As our third subject for the evaluation of our approaches, we used a total of four models from the *passenger car division* and the *truck division* of our industry partner. As these models are all confidential, we can only provide abstracted information without concrete implementation details or their original names. The two models from the passenger car division implement *versions* of an *Exterior Light Front (ELF)* of a car, which are six month apart from each other, and comprise ≈ 30,000 blocks each. The two models from the truck division implement *variants* of a *Drive Train Model (DTM)* of a truck (i.e., additional features were realized and added) and comprise ≈ 40,000 blocks each.

Together with experts from the industry partner's corresponding divisions, we partitioned these models into smaller feature submodels similar to the *DAS* case study from the *SPES_XT* project (cf. Section 7.2.2). Furthermore, the experts helped us to group logically associated feature submodels to better understand their relations. Overall, we identified three groups for the two *ELF* model versions and seven groups for the two *DTM* model variants. Abstracted details on the groups can be found in Table 7.3 and Table 7.4 for the *ELF* and *DTM* models, respectively. Along with the number of submodels per group, we show summarized details on the minimum, maximum and median size of the submodels (i.e., of their contained number of blocks), the contained subsystems and the hierarchy depths of the submodels.

Industrial – Selected Subjects for Evaluating the Family Mining Approach Similar to the *DAS* case study from the *SPES_XT* project (cf. Section 7.2.2), the main goal for this case study was to execute manual evaluation of the results to check for the *correctness* of the identified fine-grained variability information. Due to the sheer number of comparisons (29 comparisons for the *ELF* models and 98 comparisons for the *DTM* models) and the size of the compared models (up to 2,429 subsystems with up to 18,393 contained blocks) it is infeasible to execute all pairwise comparisons between the related feature subsystem versions (i.e., in case of the *ELF* models) or the related feature subsystem variants (i.e., in case of the *DTM* models) for a detailed manual analysis. Thus, we executed pairwise comparisons for 11 feature submodels of the *ELF* systems (i.e., ≈ 37.9% of all possible pairwise

Group Name	Feature Submodels	Blocks			Subsystems			Hierarchy Depth		
		min	*max*	$\tilde{b}$	*min*	*max*	$\tilde{s}$	*min*	*max*	$\tilde{h}$
AFB	11	18	1,575	230	2	201	33	1	5	2
MSS	8	37	408	197	8	53	24	1	3	2
RFL	10	29	1,922	226	3	277	33	2	4	3

$\tilde{b}$ = *median of blocks* $\tilde{s}$ = *median of subsystems* $\tilde{h}$ = *median of hierarchy depths*

Table 7.3.: Feature submodels extracted from the *Exterior Light Front (ELF)*.

Group Name	Feature Submodels	Blocks			Subsystems			Hierarchy Depth		
		min	*max*	$\tilde{b}$	*min*	*max*	$\tilde{s}$	*min*	*max*	$\tilde{h}$
AGP	13	6	821	70	1	257	22	2	4	2
ISC	26	63	18,393	583	9	2,429	75	4	6	4
OV	10	3	181	13	0	26	1	1	4	2
MLMI	11	10	24	19	0	0	0	0	0	0
MLMO	11	4	10	4	0	2	0	0	1	0
MLS	11	119	5,158	1,541	8	539	166	3	8	4
SD	16	60	3,944	209	4	339	17	2	7	3

$\tilde{b}$ = *median of blocks* $\tilde{s}$ = *median of subsystems* $\tilde{h}$ = *median of hierarchy depths*

Table 7.4.: Feature submodels extracted from the *Drive Train Model (DTM)*.

comparisons) and 22 feature submodels of the *DTM* systems (i.e., $\approx$ 22.5% of all possible pairwise comparisons) and checked the correctness of the results. In addition, these comparisons allow us to evaluate the *runtimes* and the *scalability* of our Family Mining approach in an industrial setting. In the remainder of the evaluation, we refer to these selected model variants as the *Industrial* subjects.

Overall, the selected six sets of case study subjects (three *BCS* sets, two *DAS* sets and one *Industrial* set) allow us to evaluate whether our approaches can be used for different implementation languages and project-specific settings. Thus, we are able to analyze the *adaptability* of our proposed techniques in addition to their *correctness, precision, runtime, scalability* and *usefulness*.

7.3. Methodology

In this section, we give details of our overall methodology to evaluate our approaches and answer our research questions in Section 7.1.

RQ1 – Correctness and Precision For our evaluation, we first concentrate on *RQ1* to evaluate the correctness and precision of the developed approaches:

1. *RQ1.1a & RQ1.1b* Corevid – *Cluster & Outlier Detection:* We execute the Corevid approach for the selected *BCS1* subjects and manually check the results. For our empirical evaluation of the results, we use *precision* and *recall* that are widely used in software engineering for such

evaluations [Mal15]. We refer to *precision* as the extent to which the assigned relations conform with the ground truth of the selected scenarios from the *BCS SPL*. We refer to *recall* as the extent to which the model variants have been processed and assigned a similarity in relation to the remaining model variants from the generated dendrogram.

2. *RQ1.2* Family Mining – *Level of Correctness of Results:* We execute Family Mining for the case study subjects discussed in Section 7.2 using the different implementations. For this analysis, we employ the notion of precision and recall [Mal15]. We refer to *precision* as the extent to which the assigned relations between blocks are understandable to the user and, thus, considered valid. We refer to *recall* as the extent to which model elements have been processed and assigned a relation by the Family Mining. If not otherwise stated, we execute the Execution-Flow Analysis (EFA) algorithm (cf. Section 4.4) during the evaluation of our Family Mining approach. However, to evaluate the impact of the Matching Window Analysis (MWA) algorithm (cf. Section 4.7) in contrast to the EFA algorithm, we compare their results for the *DAS* and *Industrial* subjects as hierarchy shifts and horizontal dispersions are most likely to exist for these subjects. In contrast, the *BCS* case study does not contain such complex relations due to its SPL nature (i.e., the variability is realized in a more structured manner).

3. *RQ1.3a* Matador SPL – *Correctness of Delta Languages and Modules:* Afterwards, we generate corresponding delta languages for the modeling languages used by our case study subjects and use them to migrate the analyzed variants to SPL realizations. Based on these results, we analyze whether variants derived from the SPL correspond to the variants that served as input to the variability mining process. For the manual analysis of model variants derived from the generated SPL implementations, we employ the notion of precision and recall [Mal15]. In this context, *recall* measures to what extent the derived variants comprise all necessary model elements with respect to their original variant that served as input to the Matador SPL approach. Furthermore, *precision* measures to what extent the derived variants are valid and comprise all necessary relations between these elements (e.g., the source and target state relations for transitions in statecharts).

4. *RQ1.3b* Matador SPL – *Dissection in Features:* During our evaluation of the Matador SPL approach, we not only analyze its capabilities to generate single delta modules comprising all necessary changes per variant, but also delta modules per feature. To this end, we provide the Matador SPL approach with information on the names of features contained in the analyzed *DAS* and *BCS* case study subjects and execute its feature identification facilities to generate an SPL with different features. By manually analyzing the generated SPL artifacts, we evaluate whether the Matador SPL approach is able to identify sensible features within the migrated model variants and generate corresponding features with mapped realization artifacts.

RQ2 – Runtime and Scalability We next focus on *RQ2* to evaluate the runtime and scalability of our approaches to analyze sets of model variants. To this end, we measured the runtimes during execution of the approaches for the evaluation of *RQ1* and use the following computer setups for the different parts of the evaluation:

- *Execution of the Evaluation for the Industrial Case Study*: For this part of the evaluation, we had to use a laptop provided by our industry partner as we were not allowed to copy the provided

model variants to our machine for confidentiality reasons. The provided laptop was running on Windows 7 with a 2.7 GHz Intel i5 processor and 4 GB RAM.

- *Execution of the Evaluation for the Remaining Case Studies*: For the remaining case studies, we executed our approaches on a laptop running on Windows 7 with a 2.7 GHz Intel i7 processor and 12 GB RAM.

To account for runtime deviations inherently present in a non-closed system (e.g., deviations due to scheduling between different tasks executed on an operating system), all executions to measure runtimes were performed 10 times and the average was calculated.

RQ3 – Adaptability We next focus on *RQ3* to evaluate the adaptability of our developed approaches for different settings and languages:

1. *RQ3.1* Corevid – *General Adaptability:* We discuss our experiences of adapting our Corevid approach and the underlying Samos framework for different languages and settings. This way, we not only provide details on our experiences, but also demonstrate that the Corevid approach can easily be used by companies in other contexts to provide the expected relations.

2. *RQ3.2a* Family Mining – *General Adaptability:* We adapt Family Mining for the used modeling languages of our case study subjects (i.e., IBM Rational Rhapsody statecharts and Matlab/Simulink models – cf. Section 7.2). We distinguish between the following adaptations:
   - Custom Implementation: A completely custom implementation that does not use or rely on the provided generic Family Mining facilities (e.g., the generic implementation of the algorithms), but is realized within the Family Mining framework using its generic data structures (e.g., the generic comparison element) and its processing pipeline only.
   - Manual Adaptation: An adaptation that uses the provided generic Family Mining facilities by manually implementing the corresponding extensions (e.g., for the Compare algorithms to identify the execution start nodes).
   - Vampire DSL Adaptation: An adaptation that uses Vampire DSL descriptions to semi-automatically adapt the generic Family Mining for the new languages by generating large portions of the needed facilities (e.g., the required metric).

3. *RQ3.2b* Family Mining – *Reduction of Implementation Effort:* We compare the results of the three different adaptations from *RQ3.2a* to identify whether we can achieve sensible Family Mining results using the different approaches. Furthermore, we count the *Lines of Code (LOC)* for the relevant artifacts (i.e., for *Compare*, *Match*, *Merge* and the *Metric*) of the different Family Mining realizations to quantify their implementation effort and compare them across the different implementations.

4. *RQ3.3* Matador SPL – *Language Independence:* Based on the generated variability mining results, we execute the Matador SPL approach to derive a delta language for the modeling language used to realize the corresponding model variants. Using the generated delta languages, we execute the Matador SPL approach to encode the identified variability information in corresponding SPL realization artifacts. This way, we demonstrate that the Matador SPL approach is capable of migrating model variants realized with different modeling languages

to corresponding SPLs. For the analysis of the results, we employ the notion of precision and recall [Mal15]. In case of the delta language generation, *recall* measures to what extent the required delta operations (i.e., to encode the identified variability in corresponding SPL artifacts) are generated. In this context, *precision* measures to what extent these required delta operations are generated as valid DeltaEcore delta operations. In case of the delta module generation, *recall* measures to what extent all required delta modules are generated and store the necessary delta operation calls to encode the identified variability. In this context, *precision* measures to what extent these required delta modules are generated as valid DeltaEcore delta modules storing valid delta operation calls.

RQ4 – Usefulness of Results We next focus on *RQ4* to evaluate the usefulness of the results generated by our approaches:

1. *RQ4.1* Corevid – *Improvement of Results:* Based on the identified relations, we execute our Family Mining approach by removing identified outliers and concentrating on each provided cluster. Using the corresponding results, we manually analyze whether the provided variability information is improved compared to variability information generated for the complete set of variants.

2. *RQ4.2* Family Mining – *Usefulness of Results:* We execute semi-structured interviews with domain experts within the company that provided us with the *Industrial* case study subjects. This allows us to understand what the experts' expectations are regarding variability information and how useful the details provided by our Family Mining are to such experts. For details on the interviews and the corresponding methodology, we refer to Section 7.5.

3. *RQ4.3* Matador SPL – *Usefulness of Results:* We analyze and discuss the usefulness of the SPL realizations generated by our Matador SPL approach by comparing the results for the *BCS* subjects with the ground truth of the *BCS SPL*. Furthermore, we analyze how the possibility to manually refactor the generated SPL realizations influences the corresponding evaluation.

7.4. Results and Discussion

In the following sections, we present and discuss the results for the evaluation of our research questions. To this end, we use the implementation of the Corevid, Family Mining and Matador SPL approaches, which we described in Chapter 6. As concrete subjects, we use the three case studies described in Section 7.2 and apply the methodology described in Section 7.3.

7.4.1. Results and Discussion of RQ1: Correctness and Precision

To evaluate the correctness and precision of our approaches, we performed a detailed manual analysis for the execution results of the selected case study subjects.

RQ1.1a Corevid – Outlier Detection For each of the selected outlier detection scenarios `OD1` to `OD3` from the *BCS1* subjects, we executed the Corevid approach to generate dendrograms depicting possible clusters and outliers. In Figure 7.1a to 7.1c, we present the corresponding dendrograms. During a manual analysis of the results, we interpreted the dendrograms and added frames to highlight the identified clusters and outliers. The interpretation of the results is as follows:

- *Clusters:* Big coherent groups of model variants with a high similarity are regarded as clusters and marked using blue frames.
- *Outliers:* Individual model variants with a relatively small similarity compared to the main cluster are regarded as *outliers* and marked using red frames.

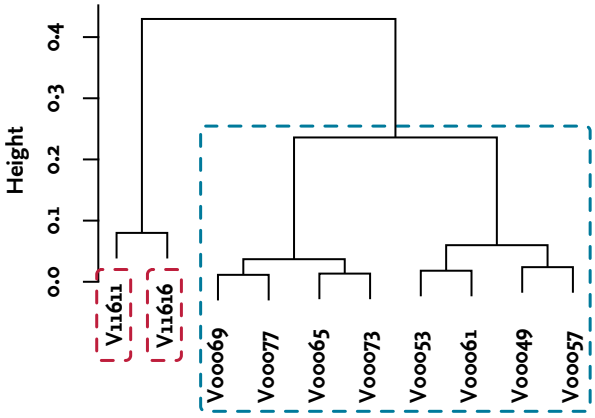
Based on these annotations, we compared the results with the ground truth of the *BCS SPL*. During this analysis, we identified that the Corevid approach exhibits total precision and recall for these scenarios as it assigned a similarity to each variant and generated results conforming with the ground truth.

An important point to discuss is related to the distinction between a feature vs. its implementation in the variant models. While we designed the scenarios based on selecting/deselecting features to comprise a notion of similarity between models, we ignored their implementation, especially how big their corresponding realizations in the models are. This may lead to some situations where a common feature with a very large implementation offsets the selection of all other minor features and dominates the similarity calculation of the Corevid approach. This is partly reflected in Figure 7.1c, where a considerable number of different features (6 out of 27) between `V11616` and the large cluster contribute only to around 10% difference. Elaborate weighting schemes to account for such situations are not further investigated within the given scope of this thesis. For example, in future work it could be worth investigating such schemes to weight the importance of features or model elements higher. Nevertheless, the Corevid algorithm is able to correctly identify sensible clusters conforming to the ground truth of the *BCS SPL* (i.e., similarity of the variants based on their selected features). Thus, we answer *RQ1.1a* positively as the Corevid approach is able to indicate such clusters and outliers with sufficient accuracy with respect to the expected relations.

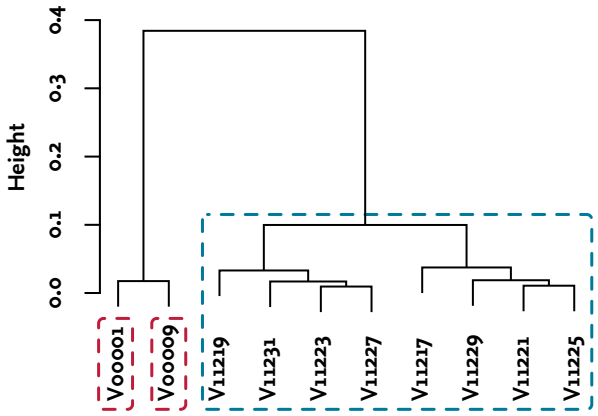
RQ1.1b Corevid – Cluster Detection For the cluster detection, we adopted an approach similar to the outlier detection. However, this time we tried to find large groups of similar models rather than isolated outliers during the detailed analysis of the dendrograms generated by the Corevid approach. Looking at the dendrograms in Figure 7.2a and Figure 7.2b, we can clearly see that for both scenarios there are two distinct sets of model variants with high similarity among each other (i.e., with a small distance of about 0.1 and 0.2, respectively) and an intra-cluster distance of between 0.25 and 0.3 that is high enough to comprise separate groups. As mentioned previously, especially in the case where a much larger number of models is considered, it is arguable whether to obtain a few but large clusters, or many but small (sub-)clusters. Thus, we identified that the Corevid approach also exhibits total precision and recall for these scenarios as it assigned a similarity to each variant and generated results conforming with the ground truth.

Altogether, the results of the outlier and cluster detection scenarios confirm that our Corevid approach is able to perform with sufficient accuracy with respect to the ground truth of the *BCS SPL* for this study. Thus, we overall answer *RQ1.1b* positively.

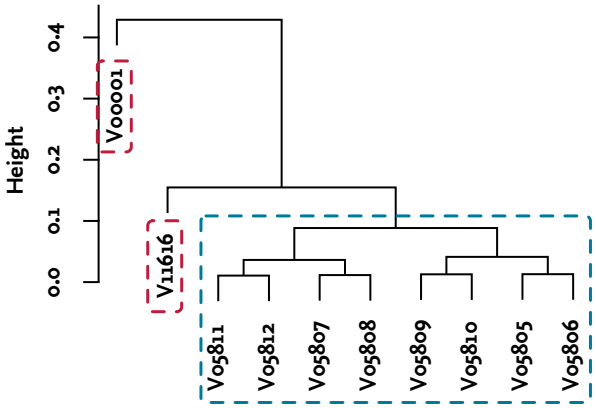
RQ1.2 Family Mining – Level of Correctness of Results To evaluate the level of correctness of our results, we *directly* evaluated 33 model comparisons from the *Industrial* subjects (i.e., 11 *ELF* and 22 *DTM* model comparisons), 51 model comparisons from the *DAS1* subjects and 18 model comparisons from the *BCS2* subjects by manually analyzing their Family Mining results. Furthermore, we *indirectly* evaluated 7 model comparisons from the *BCS1* subjects comprising between 8 and 12 compared models as well as 8 model comparisons from the *DAS2* subjects by executing the Matador SPL



(a) Dendrogram for OD1.

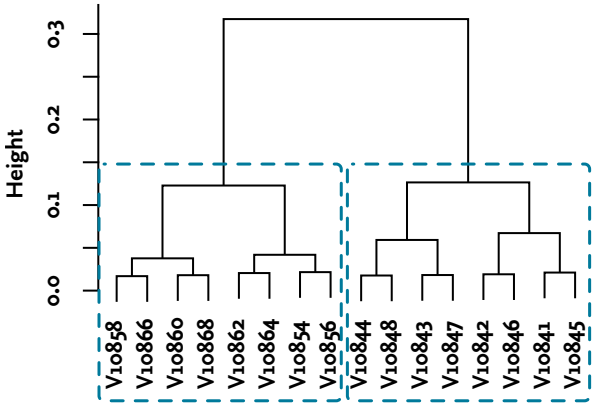


(b) Dendrogram for OD2.

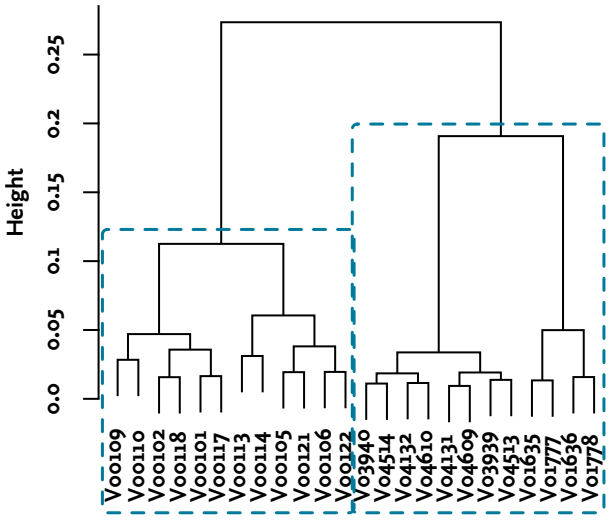


(c) Dendrogram for OD3.

Figure 7.1.: Dendrograms generated by the CoREVID approach for the outlier detection scenarios OD1 to OD3 from the *BCS1* subjects. We manually marked identified clusters with blue frames (larger groups) and outliers with red frames (single variants).



(a) Dendrogram for CD1.



(b) Dendrogram for CD2.

Figure 7.2.: Dendrograms generated by the CoREVID approach for the cluster detection scenarios CD1 and CD2 from the *BCS1* subjects. We manually marked identified clusters with blue frames.

approach for them and evaluating the products generated by the derived SPL realization (cf. discussion of results for *RQ1.3a* in Section 7.4.1). Thus, any problems with the generated products of the derived SPL would indicate potential errors in the Family Mining as the corresponding results represent the basis for the Matador SPL approach.

When evaluating the results for the analyzed subjects, we at first found that some errors were present in the 150% models generated by the Family Mining approach executed for them. During a detailed analysis two problems became apparent:

1. *Missing Variability Annotations:* For the Matlab/Simulink case study subjects (i.e., the *Industrial* and *DAS1* subjects), we found minor problems in the merging algorithms that resulted in inaccurate annotation of the identified variability. For example, in a few cases the annotation about the containing models for mandatory model elements was missing the information about the second model. In addition, we found that in certain cases the variability and information of merged connectors was incorrect. Further investigating these issues, we found that our Family Mining algorithm identified the variability correctly, but the merging algorithm did not correctly process the information during generation of the 150% models.

2. *Incorrect Variability Relations for Regions:* For the statecharts from the *BCS2* subjects, we identified that in certain cases the implemented merging did not correctly identify optional regions and rather represented them as alternatives. A detailed discussion of this problem and its solution can be found in Section 4.6.2.

After fixing the identified errors in the merging of the Family Mining approach for the statechart and Matlab/Simulink implementations, we executed the approach again for the corresponding model comparisons. During a manual evaluation of these results, we found that fixing these minor bugs in our implementation indeed solved the identified problems. Lesson learned for us is that a structured and focused implementation of the merging algorithms is essential to generate correct variability relations. In this context, we found that paying attention to the variability relations expected by users is crucial to identify incorrect results and rectifying them. This is most important in cases where the generated results are further processed by approaches relying on their correctness (e.g., the Matador SPL approach). Thus, developers should spend enough time on testing their realization (e.g., with realistic scenarios where the expected results are known upfront) prior to any productive application by experts.

When analyzing the precision and recall for the selected case study subjects, we found for the *BCS2* subjects that all identified variability conforms to the ground truth of the *BCS SPL* and, thus, our Family Mining approach exhibits a total precision for them. Furthermore, we identified that the Family Mining has a total recall for the evaluated subjects as all compared model elements are contained in the generated 150% model and have a variability relation assigned to them.

During the evaluation of precision and recall for the *DAS1* and *Industrial* subjects, we executed both the MWA and EFA Family Mining algorithms. For the *DAS* subjects, we identified that all analyzed results conform with the variability expected by the executing experts and, thus, we argue that the Family Mining exhibits total precision for the selected subjects. Furthermore, we were able to support our findings regarding the recall as all compared model elements were contained in the generated 150% models and had a variability relation assigned to them.

For the evaluation of precision and recall for the *Industrial* subjects, we found that differences exist between the results of the MWA and EFA Family Mining algorithms. Three of the results generated by the EFA algorithm for the *ELF* models were identified to be incorrect when manually analyzing them in detail. In contrast, the MWA algorithm was able to correctly identify the hierarchy shifts and horizontal dispersions causing the invalid results and, thus, we were able to validate them. Thus, depending on the use case at hand (i.e., whether hierarchy shifts and horizontal dispersions are present in the analyzed models), it can make sense to execute the MWA algorithm instead of the EFA algorithm. However, as the MWA algorithm has longer runtimes due to its higher complexity (cf. the detailed discussion on *RQ2.2* in Section 7.4.2), we do not per se recommend applying it for every use case. For the results generated for the *DTM* models, we found all manually evaluated combinations to be correct for both algorithms. When discussing these findings with the experts from our industry partner, they confirmed that in contrast to the *ELF* models no hierarchical shifts or horizontal dispersions were present in the *DTM* models. This is due to the nature of the provided *DTM* models as they represent variants of a system and, thus, are less likely to contain these peculiarities due to the structured design of variability for such variants at our industry partner (cf. Section 7.5). In contrast, the found *ELF* models represent versions of a system, where such hierarchical shifts or horizontal dispersions might be present at our industry partner when introducing hotfixes prior to a major release (cf. Section 7.5).

To summarize the findings for the *Industrial* subjects, the EFA algorithm might not identify valid results in all situations (i.e., when hierarchy shifts or horizontal dispersions are present). However, the MWA algorithm is able to account for these inaccuracies. We identified that the Family Mining approach exhibits total precision for the selected subjects. Furthermore, it has a total recall for the analyzed subjects as we found that all compared model elements were contained in the generated 150% models and had a variability relation assigned to them.

As the *Industrial ELF* and *DTM* subjects represent *variants* and *versions* of systems (cf. Section 7.2.3), our evaluation of the algorithms' correctness shows that the Family Mining algorithms are applicable in both cases. Thus, we are confident that our approach can be used to provide relations between models not only in explicit variability scenarios, but also in evolutionary scenarios.

Overall, we can summarize that our Family Mining exhibits a total precision and total recall for all selected subjects. Thus, we can answer *RQ1.2* positively as the Family Mining approach is able to find precise variability information even in industrial settings with large models present.

RQ1.3a Matador SPL – Correctness of Delta Languages and Modules Automatically migrating a set of variants to an SPL requires that the generated SPL artifacts allow at least generation of the variants that served as input. Furthermore, the generated product variants should conform to the variability of the SPL and represent valid variants (i.e., their implementation is correct).

To this end, we generated delta languages only comprising the necessary delta operations for the modeling languages of the model variants that were migrated to corresponding SPL realizations. Afterwards, we derived all input variants from the SPLs generated by the Matador SPL approach for the *BCS3* and *DAS2* subjects. When analyzing the generated variants, we found that they exhibit total precision and recall as all variants conformed with the respective variants that served as input to the Matador SPL migrations. During additional analysis, we found that the created SPLs allow to derive additional variants over the migrated input variants. Each of these variants, was manually analyzed by us and found to represent a valid variant. As a result, we explicitly highlight that

our Matador SPL approach does not only generate SPL artifacts that are capable of generating the initial input variants, but also of providing previously unavailable new variants. Overall, this direct benefit gives additional motivation for executing our Matador SPL approach as requirements by potential new customers can easily be satisfied without additional implementation effort and investing in this solution can pay off in the short-run.

Overall, we answer *RQ1.3a* positively as our Matador SPL approach allows to generate delta languages exactly comprising the delta operations that are required to migrate a set of variants to corresponding SPL realizations. Furthermore, the SPL realizations generated by the Matador SPL not only allow to derive the input variants, but also allow direct derivation of additional variants from such an SPL.

RQ1.3b Matador SPL – Dissection in Features To generate a highly configurable SPL realization allowing for derivation of different product variants it is essential to dissect the identified variability into different features with associated realization artifacts. During our evaluation of the Matador SPL approach, we did not only analyze its capabilities of generating single delta modules comprising all necessary changes per variant, but also delta modules per feature.

During the manual analysis of the Matador SPL results generated for the provided feature information, we identified:

1. *Sensible Features:* The features generated by the rather naive approach of searching for model elements with a specific name in the model hierarchy (cf. Section 5.4) is able to derive sensible features for the analyzed case studies. For each of the feature models derived by the Matador SPL approach, we observed that the features found for the evaluated *DAS2* and *BCS3* subjects were understandable. Furthermore, the identified features were linked with matching and fully functional realization artifacts in form of generated delta modules. Upon closer inspection of these delta modules, we also found that the contained delta operation calls were referring to matching model elements implementing the features in the input variants. In Section 5.4, we showed corresponding exemplary results for the dissection of *BCS* variants into features together with a corresponding feature model in Figure 5.2. During a subsequent refactoring of the created SPL, we were able to restructure the SPL artifacts (cf. the feature model in Figure 5.4) and to create a realization that is close to the ground truth of the original *BCS* feature model (cf. Figure 2.9). As the restructured solutions are close to the ground truth of a realization that was created by domain experts using SPL best practices, we are confident that our approach is able to actually identify sensible features. Overall, our impression is that the close coupling between the implementation of the input variants and the corresponding feature model is an advantage as the relations become easily apparent and help during subsequent analysis and refactoring of the generated SPL.

2. *Sensible Merging of Features:* During the generation of the SPL artifacts, we compare the delta operations of features that are identified in multiple input variants and merge corresponding feature implementations with equal operations into a single feature (cf. Section 5.4). The goal of this approach is to reduce the number of features in the generated feature model, because, otherwise, the SPL would comprise a feature for each input variant implementing it. While we at first were not sure, whether this approach is actually able to reduce the number of features, we were positively surprised of its effects. For the evaluated results from the *DAS2* and *BCS3*

subjects, we observed that in most cases the number of features were significantly reduced and mostly merged into a single implementation. Upon closer investigation of all other cases, we identified that only up to three implementations were generated for the case study subjects (for the larger part only two), which actually differ due to specific feature implementations in the variants. Thus, we overall identified the merged feature implementations to be correct.

3. *Feature Interactions:* With respect to these merged features, we found that the differing implementations leading to multiple realizations of a feature across the variants are mostly due to interactions of the feature with other features. For instance, depending on the selected Power Window feature (either AutoPW or ManPW) of the *BCS* case study, the Finger Protection (FP) feature is implemented differently. While the larger part of the FP feature uses the same functionality, minor differences exist in the used transition actions resulting in differing implementations. During additional experiments, we tried to further identify the common parts between the features and extract them as a shared delta module. However, we found that automatically identifying feature interactions and finding a generalizable solution that is able to extract them for different use cases is not a trivial task. Thus, a possible solution for future work could be to exploit additional user-provided knowledge (e.g., a seed fragment, such as a relevant transition action in our example) to identify and extract such interactions. For such additional algorithms, our current solution provides a sensible starting point as the dissected features allow derivation of valid variants and correctly separate the differing implementations of features.

Altogether, we identified the features and corresponding delta modules generated by the Matador SPL approach to be correct for the analyzed subjects. Thus, we overall answer *RQ1.3b* positively.

7.4.2. Results and Discussion of RQ2: Runtime and Scalability

To get a detailed understanding of our algorithms' runtimes for realistic applications, we measured runtimes during their execution for our selected case study subjects. Based on this information, we also execute a scalability analysis to understand the algorithms' performance.

RQ2.1 Corevid – Runtime and Scalability To evaluate the runtime of our Corevid approach, we executed it for the *BCS1* subjects to identify the clusters and outliers in these model variants. In Table 7.5, we present the corresponding runtimes, together with the runtimes of the Family Mining approach executed for the identified clusters and a migration to an SPL using the Matador SPL approach. As we can see, executing the complete pipeline of the Corevid approach, Family Mining approach and Matador SPL approach takes between about 30 and 60 seconds for the analyzed model variants. In addition, we have to consider the required time for adapting each language for the Family Mining. As the required time to manually adapt Family Mining for a new language is highly dependent on a large variety of factors (e.g., the developer's experience and the language's complexity), it is hard to give accurate measures for this effort. Thus, we concentrate on the adaptation times required to execute the Vampire DSL generator for a Vampire DSL description of the corresponding languages including their meta-model, a matching metric and variability thresholds. For Matlab/Simulink, the generation of all adaptation artifacts required about 4,604.7 ms and for statecharts adaptation about 4,435.5 ms. These runtimes do not include manually writing the Vampire DSL (as this requires analyzes of the used language elements), implementation of the

		Corevid	Family Mining			Matador SPL		Overall
			Compare	*Match*	*Merge*	*Dialect*	*SPL*	
OD1		23,631.3	3,948.6	256.2	265.0	125.6	544.2	28,770.9
OD2		27,882.5	17,243.6	973.8	569.5	149.8	491.6	47,310.8
OD3		33,268.7	14,455.8	690.4	508.9	184.2	509.6	49,617.6
CD1	*Cl. 1*	47,866.3	9,810.5	1,102.5	755.3	165.3	882.6	60,582.5
	Cl. 2		9,785.9	939.3	536.8	141.6	753.1	60,023.0
CD2	*Cl. 1*	37,581.1	7,919.8	639.4	468.6	144.3	325.4	47,078.6
	Cl. 2		8,497.4	914.1	501.1	132.2	621.6	48,247.5

Table 7.5.: Average execution times (in milliseconds) of the Corevid approach, the generic Family Mining approach and the Matador SPL approach for the *BCS1* subjects selected in Table 7.1.

merge algorithm (as this requires understanding of the expected 150% model) and writing possibly needed manual extensions to the generated code (e.g., to identify execution start nodes).

When analyzing the percentage that each executed approach takes in relation to the overall runtime, we can see that 66% of the overall runtime are required by the Corevid approach. In contrast, the Family Mining approach and the Matador SPL approach account for only 32% and 2%, respectively. While this indicates that the Corevid approach is major contributor to the overall runtime, we have to keep in mind that the Corevid approach is a) optional in cases where the relations between models are already known and b) has only to be executed once per set of model variants.

As the models from the *BCS* case study comprise only a limited set of model elements (i.e., up to 283 regions, states and transitions per variant), the presented results only give an overview of possible runtimes and we cannot give precise information on the scalability of the approach. However, when looking on a detailed evaluation of the underlying Samos framework by Babur et al. [BCB18], we can see that experiments with similar settings (i.e., using typed bigrams) for clustering large sets of meta-models (i.e., 250 meta-models with about 50,000 model elements) required about four hours with caching of word-to-word similarities and about 14 hours without such caching, respectively. To further improve these runtimes, Babur et al. [BCB18] extend their approach using distributed cloud computing based on Apache Spark[3] and Apache Hadoop[4]. The evaluation of the distributed implementation showed that with an increasing number of distributed executors the runtime can largely be reduced. For the used data set of 250 meta-model, 50 executors with seven processor cores and 8 GB RAM each were sufficient to outperform the single system solution with activated caching as only about three hours were required to achieve the same results even when deactivating caching. Obviously this solution requires additional effort to setup the distributed execution, but is not completely unrealistic given the server infrastructure of many modern companies.

Furthermore, considering that the Corevid approach only needs to be executed *once* in cases where the *relations are unknown*, we argue that such runtimes *with* or *without* distributed execution are very well acceptable. Thus, we answer *RQ2.1* positively.

[3] https://spark.apache.org/
[4] https://hadoop.apache.org/

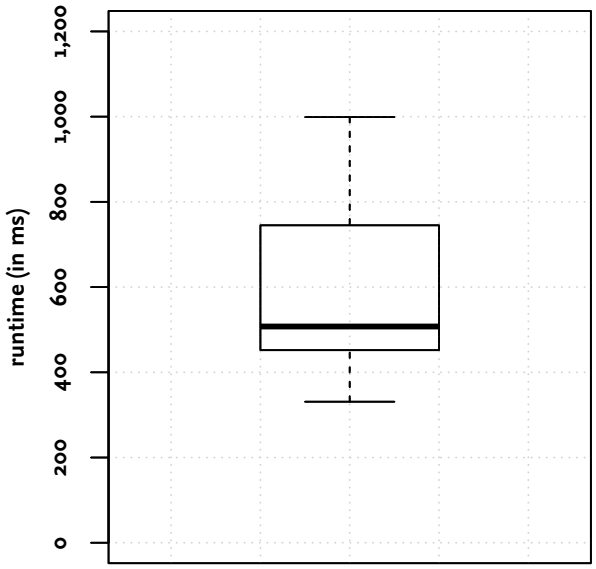


Figure 7.3.: Boxplot of the runtimes for executing the Family Mining approach for the *BCS2* subjects.

RQ2.2 Family Mining – Runtime and Scalability In Figure 7.3, we present a boxplot for the execution of our Family Mining approach for the selected 27 scenarios of the *BCS2* subjects. The runtime ranges between 331 ms (184 compared model elements) and 999 ms (468 compared model elements), whereas the median runtime for these subjects is about 500 ms. Thus, the runtimes of the Family Mining approach for these rather small, but realistic, model variants from the *BCS* case study are within a very acceptable range. Furthermore, we argue that the Family Mining approach would outperform any manual variability analysis of the same models even with longer runtimes.

In addition to the runtimes, we were interested in the scalability of the approach and set the Family Mining runtimes in relation to the average number of model elements involved in the corresponding comparisons. In Figure 7.4, we show the corresponding scatterplot with a trend line to visualize the algorithm's behavior for growing numbers of compared model elements. We can see that the data points are scattered more or less evenly around the trend line and 80% of all markers are in an interval of $\pm$100 ms around it. Overall, the created trend line gives the impression of an exponential trend. However, the considered window with only up to an average number of 234 compared model elements is too small to give a precise answer. Thus, with the *BCS* case study being limited to rather small model variants, we further evaluated the runtimes and scalability of the Family Mining approach with the *DAS1* and *Industrial* subjects.

In Figure 7.5, we show boxplots for the runtimes of the *DAS1* subjects showing the number of feature submodels that were involved in the corresponding comparisons in relation to the runtime of the algorithms. In addition to the EFA algorithm (*left* boxplot in each category), we executed the MWA algorithm (*right* boxplot in each category) for the selected *DAS1* subjects to better understand their relations with respect to the runtimes.

As we can see, both algorithms take only seconds to process the input models of the *DAS1* subjects. For the executed comparisons, the MWA algorithm takes up to about 5.5 seconds to identify the relations for the largest models and requires on average about 148% more time than the EFA algorithm. The presented data suggests a quadratic increase in runtime for a growing model size.

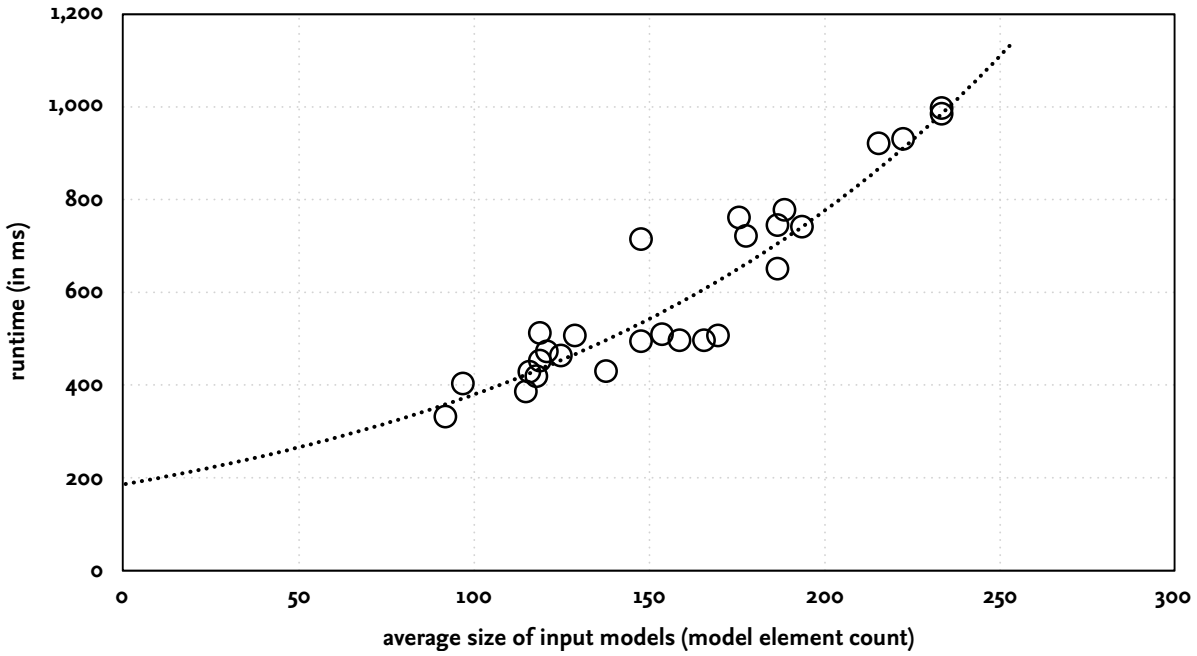


Figure 7.4.: Family Mining runtime in relation to the model sizes of the *BCS2* subjects.

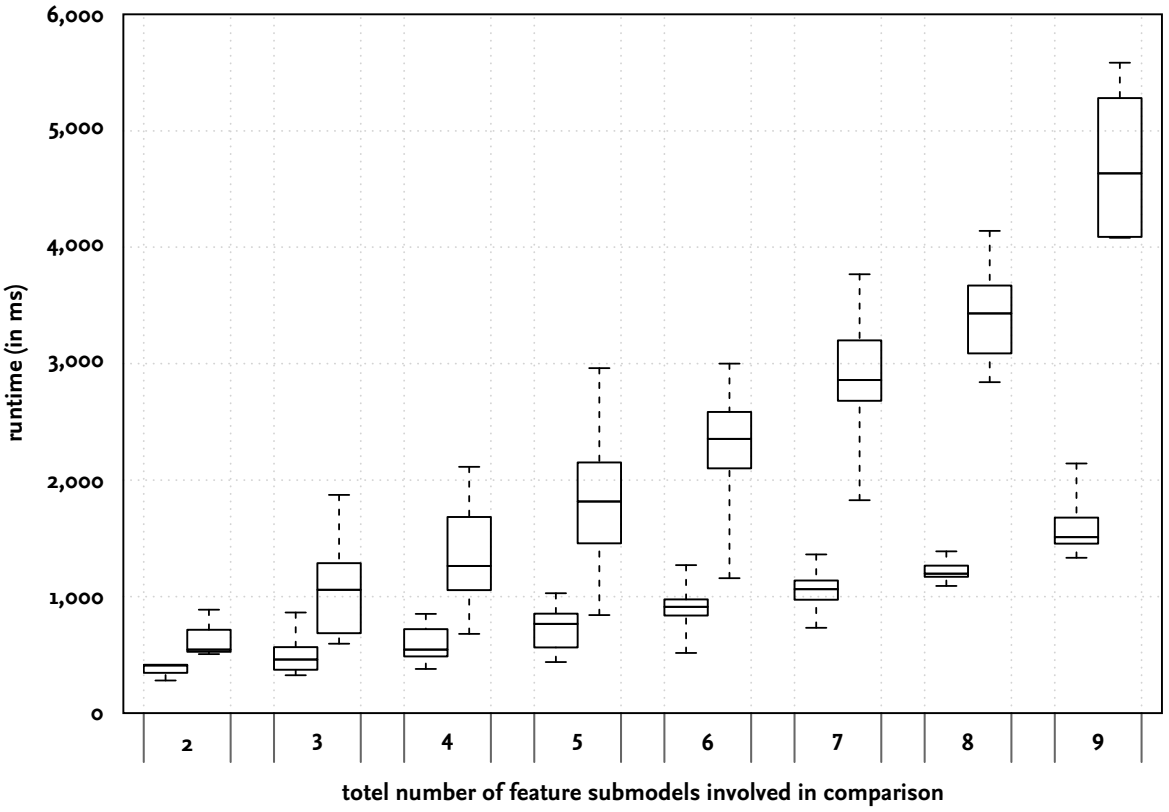


Figure 7.5.: Boxplots of the runtimes for executing the Family Mining approach for the selected *DAS1* subjects in relation to the involved feature submodels (*left:* EFA algorithm / *right:* MWA algorithm).

For the evaluated scenarios, outliers are more noticeable for the MWA algorithm than for the EFA algorithm. We identified that the main reason for this behavior is the impact of the models' complexity on the window generation process, which varies across the processed models.

To further investigate, whether we can confirm our findings for models in a realistic industrial setting, we investigated the *Industrial ELF* and *DTM* subjects. In Figure 7.6 and Figure 7.7, we show plots for the average model size of the analyzed *ELF* and *DTM* subjects in relation to the runtime of the executed EFA and MWA algorithms. Furthermore, we added trend lines to visualize the algorithms' behavior for growing numbers of compared model elements and to make differences between the runtimes of the algorithms more apparent. The *x*-axis uses a logarithmic scale to make the clouds of data points close to the origin of the diagrams more visible.

As we can see, the trend lines in both diagrams indicate a quadratic increase in runtime for the two algorithms, which coincides with our findings for the *DAS1* subjects in Figure 7.5. With about 75%, the *Compare* and *Match* phases account for the majority of runtime, while the *Import* phase accounts for about 8%, the *Merge* phase for about 12% and the *Export* phase for about 5%. Considering the correlation between the total size of the input models and a runtime that is within seconds, we are confident that our techniques' performance is acceptable for real-world models.

In addition, we measured the number of comparison elements that were created during the execution of the MWA and EFA Family Mining algorithms for the *Industrial ELF* and *DTM* models. In Figure 7.8 and Figure 7.9, we show the corresponding plots with added trendlines. In contrast to the previous plots on the algorithms' runtime scalability, these plots use a linear scale for their *x*-axis.

Looking at these plots, we can see that the quadratic trend in the runtime of the Family Mining coincides with the quadratic increase of the comparison elements generated by the Family Mining algorithms for growing sizes. Thus, these results support the scalability analysis of the algorithms' runtime. Furthermore, they support our finding during the execution of the evaluation that the memory space required for storing all comparison elements is the major limiting factor for scalability, rather than the runtime itself. Specifically, the industrial models exhibit a strong vertical design with a low number of stages but each stage comprising a multitude of blocks. With the EFA algorithm comparing all blocks within a stage, the number of comparisons increases quadratically. With multiple hierarchical layers and all comparison elements being stored when descending into model hierarchy, the MWA algorithm's sliding window approach amplifies this fact even further and overexerts the 4 GB of RAM available for the *Industrial* case study subjects.

When comparing the number of comparison elements generated by our EFA and MWA Family Mining algorithms with a naive algorithm comparing each model element from one model with all model elements from the other model, we can further see that our Family Mining algorithm actually reduces them by a significant number. In Table 7.6, we show such a comparison for all *ELF* and *DTM* models in Figure 7.8 and Figure 7.9. For the naive algorithm, we assume a simple multiplication of the compared model sizes[5], while we use the comparison element numbers measured during our executions for the MWA and EFA algorithms. As we can see, the average reduction of comparisons in relation to a naive algorithm is about 72.5% for the EFA algorithm and about 62.6% for the MWA algorithm when executing them for the *ELF* and *DTM* models. The slightly lower number for the MWA algorithm can be explained with the larger number of comparison elements for this approach due to the analysis of hierarchy levels and the applied window comparisons.

[5]For example, 248 × 351 = 87,048 comparisons for two models of size 248 and 351, respectively.

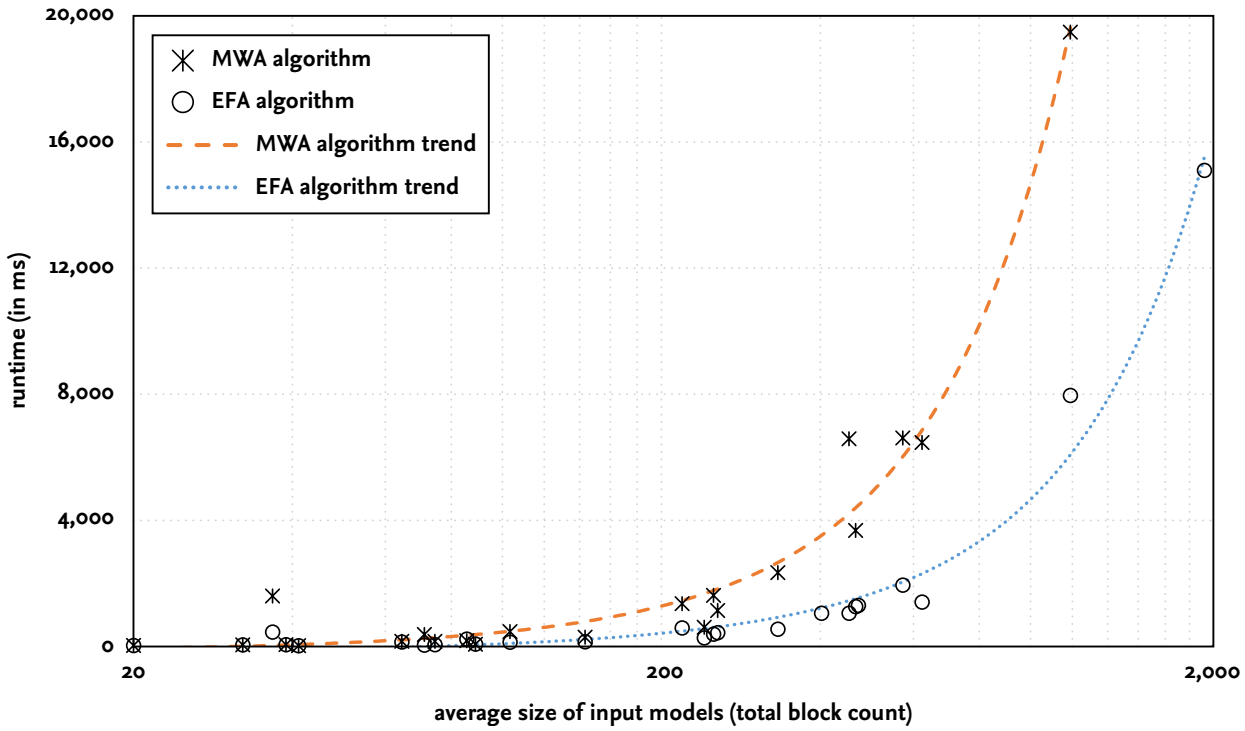


Figure 7.6.: Runtimes for the EFA and MWA Family Mining algorithms in relation to a logarithmic representation of the model sizes for the *Industrial ELF* models. In addition, we added quadratic regression fits to indicate the trend for models with growing numbers of model elements.

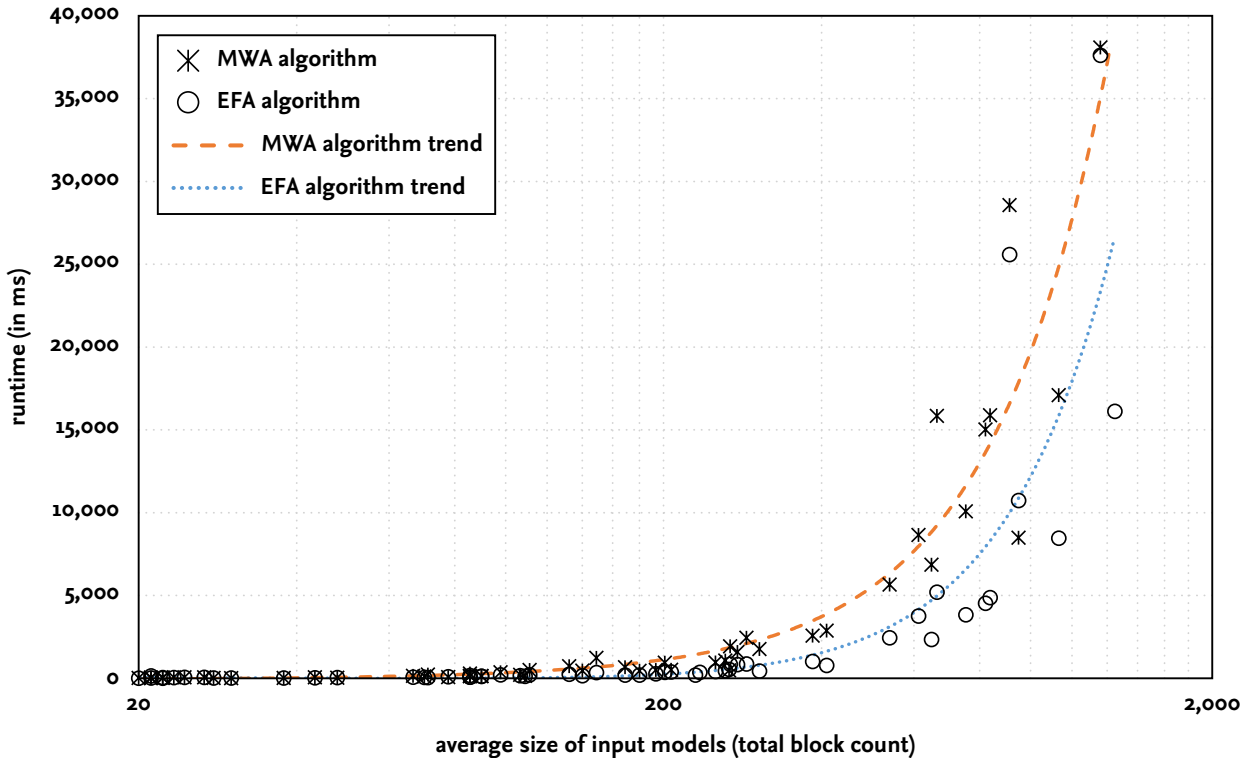


Figure 7.7.: Runtimes for the EFA and MWA Family Mining algorithms in relation to a logarithmic representation of the model sizes for the *Industrial DTM* models. In addition, we added quadratic regression fits to indicate the trend for models with growing numbers of model elements.

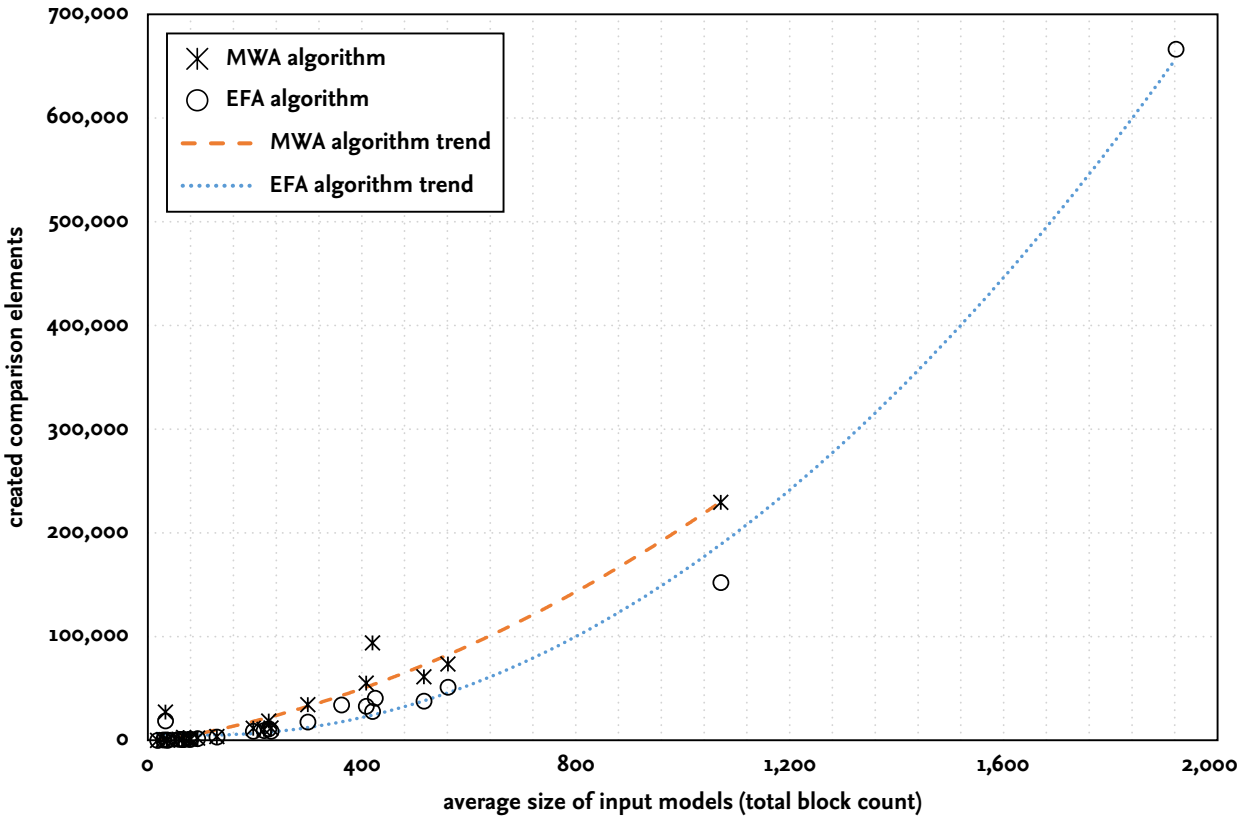


Figure 7.8.: Number of comparison elements generated by the EFA and MWA Family Mining algorithms in relation to the model sizes of the *Industrial ELF* models. In addition, we added quadratic regression fits to indicate the trend for models with growing numbers of model elements.

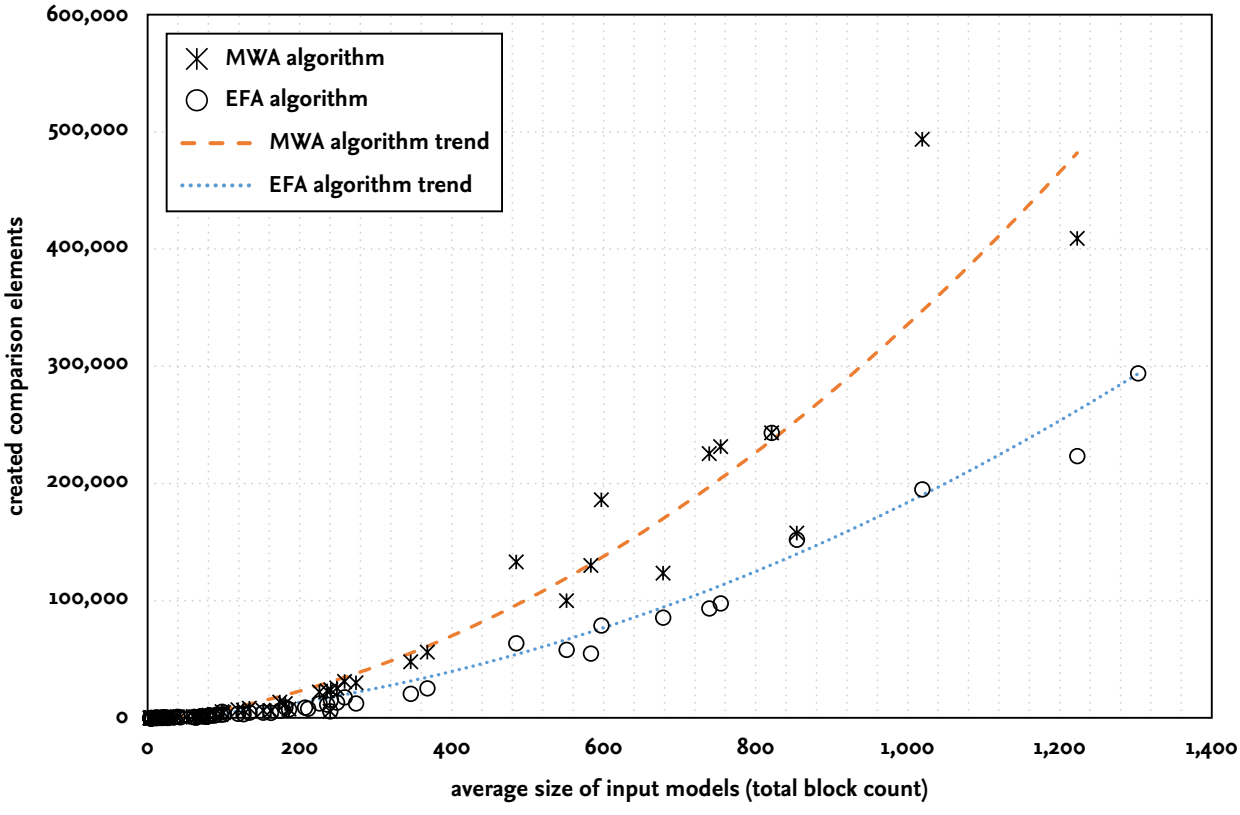


Figure 7.9.: Number of comparison elements generated by the EFA and MWA Family Mining algorithms in relation to the model sizes of the *Industrial DTM* models. In addition, we added quadratic regression fits to indicate the trend for models with growing numbers of model elements.

	ELF		*DTM*	
	EFA	*MWA*	*EFA*	*MWA*
Minimal Reduction in %	−50.89%	−16.41%	−15.63%	−15.63%
Maximum Reduction in %	−86.74%	−84.41%	−95.01%	−95.01%
Average Reduction in %	−78.32%	−67.28%	−66.79%	−58.00%
Median Reduction in %	−80.00%	−74.58%	−73.56%	−61.00%

Table 7.6.: Exemplary comparison of the executed comparisons for the EFA and MWA algorithm in relation to a naive algorithm for the *ELF* and *DTM* models.

Furthermore, the low value for the minimal reduction of comparison elements can be explained with the effect of small models on the algorithms. These models often comprised only about 10 to 50 model elements with a small number of stages comprising most of the blocks. As a result, the EFA and MWA algorithms behave similarly to a the naive algorithm as they cannot exploit the advantages of processing the models in multiple small stages comprising only few model elements, but rather have to process them in a few big stages comprising large sets of model elements. Overall, we can see from the numbers in Table 7.6 that our Family Mining algorithms are able to realize quite efficient identification of variability information for models.

In summary, we found the runtimes of the Family Mining approach to exhibit sensible runtimes for the identification of correct variability information (cf. Section 7.4.1) in realistic industrial models. Furthermore, we found the runtimes as well as the generated comparison elements to follow a quadratic trend for large models, allowing scalability for industrial-scale scenarios. These results were confirmed for the EFA as well as the MWA algorithms, allowing for a fast and precise variability analysis in different settings (i.e., with and without hierarchy shifts or horizontal dispersions present). Thus, the Family Mining approach is able to outperform any manual identification of similar fine-grained variability relations as it provides the expected results even for large models within seconds. In conclusion, we answer *RQ2.2* positively.

RQ2.3 Matador SPL – Runtime and Scalability To evaluate the runtime of our Matador SPL approach, we executed it for the *DAS2* and *BCS3* subjects to identify the clusters and outliers in these model variants. For the *DAS2* subjects the average runtime to export the delta modules was about 856.86 ms (minimum: 135.7 ms, maximum: 2,581.9 ms, median: 396.95 ms) and accounted for about 33.2% of the overall average runtime of 2,341.14 ms (including the needed Family Mining). For the *BCS2* subjects (i.e., the second set of subjects in *BCS3*) the average runtime to export the delta modules was about 57.46 ms (minimum: 11 ms, maximum: 157.5 ms, median: 55.4 ms) and accounted for about 6.1% of the overall average runtime of 943.91 ms (including the needed Family Mining). For the *BCS1* subjects (i.e., the first set of subjects in *BCS3*), we already presented the runtimes in Table 7.5. For these subjects, we observed that the runtime of executing the Matador SPL approach accounts for about 6% of the overall runtime (about 1.2% for deriving the delta dialect and about 4.8% for generating the SPL), when neglecting the onetime effort of identifying clusters and outliers.

These numbers show that even for larger models, such as the *DAS2* subjects, the Matador SPL approach is able to migrate the identified variability information to an SPL within very well accept-

able runtimes. Furthermore, we argue that even much longer runtimes in the magnitude of hours would be acceptable as a migration to an SPL is not executed on a daily basis by companies. In addition, such a migration could even be executed overnight on a company server with much more execution power than the off-the-shelf consumer laptop used for our evaluation. Thus, we answer *RQ2.3* positively with respect to the Matador SPL approach.

Overall, we showed that the approaches presented in this thesis have sensible runtimes that are well within an acceptable range. Furthermore, we showed that the presented techniques exhibit scalability for realistic scenarios. We argue that the Corevid approach will most likely not be executed on a daily basis in a company as identifying clusters and removing outliers is only necessary in cases where such details are lost (e.g., after acquiring another company and integrating its models into the own repository). Thus, after restoring this information in the common model repository, developers can easily maintain it during their everyday work. Similarly, the Matador SPL approach is only executed when companies decide to migrate their products (or a subset to an SPL). Thus, the most critical approach regarding runtime is the Family Mining approach as it might not only be used for the migration to an SPL, but also during everyday work when analyzing models (e.g., when fixing bugs in a set of models). Especially, when considering these details, the shown runtimes are well within an acceptable range as the Family Mining approach is able to identify variability information even for industrial-scale models within seconds (cf. discussion of results for *RQ2.2* in Section 7.4.2). Thus, we overall answer *RQ2* positively.

7.4.3. Results and Discussion of RQ3: Adaptability

As adaptability for different settings and languages is one of the key aspects in this thesis, we analyze the corresponding capabilities for all developed approaches.

RQ3.1 Corevid – General Adaptability During the evaluation, we showed the adaptability of the Corevid approach only for the statechart variants from the *BCS SPL*. Main reason is that finding realistic and sufficiently large case studies with a corresponding ground truth (i.e., clusters and outliers) is hard. While this limits the generalizability regarding the adaptability of our Corevid approach, we are confident that the described algorithms are applicable to other scenarios as we were successfully able to perform experiments for other languages. For instance, during our research on identifying variability relations in technical architectures (cf. [WWS+17c]), we extended the Samos framework by Babur et al. [Bab16, BCB16, BCV+16, BC17, BCB18] with a different extraction scheme and analyzed the high-level relations for these models. However, due a change of the focus in the corresponding research project, we did not further follow this direction, although, the results were promising and showed the expected relations. Furthermore, Babur et al. [Bab16, BCB16, BCV+16, BC17, BCB18] demonstrated applicability and, most importantly, adaptability of their Samos framework for different (meta-)models during their research using different extraction schemes. Thus, we overall answer *RQ3.1* positively as the underlying Samos framework of the Corevid approach was specifically designed for adaptation to different settings and languages relying on the Ecore meta-model. Furthermore, the approach proved these capabilities in different scenarios [Bab16, BCB16, BCV+16, BC17, BCB18] and, thus, should be easily applicable for new languages.

RQ3.2a Family Mining – General Adaptability During this evaluation step, we found that, in general, we were able to adapt Family Mining through the three considered strategies (i.e., Custom Implementation, Manual Adaptation and Vampire DSL Adaptation). Furthermore, we found that the

variability information identified by these different Family Mining realizations provide the same rich variability information *when* following our guidelines discussed in Chapter 4. In the following, we discuss our experience during the executed adaptations. For this discussion, we follow the four steps of our guidelines as a general structure (cf. Chapter 4):

1. *Analyzing the Block-Based Language:* For this step, we found that the available resources and the quality of their provided details has a high impact on the subsequent steps. In case of Family Mining for Matlab/Simulink models, we had access to engineers from our industry partner with years of experience in developing software using the analyzed language. Thus, we were able to combine information from the Matlab/Simulink documentation with well-founded details from additional discussions with these experts. In contrast, adapting Family Mining for statecharts proved to be more difficult. First, when we realized that a large number of different statechart dialects exists in industry, we decided that considering multiple dialects would increase the generalizability of our approach. Second, we did not have access to developers to gather additional information, but had to fully rely on the documentation and examples available. Thus, we comprehensively reviewed the available documentation and aligned the concepts of the considered dialects. Overall, this step follows the same approach for the three adaptation strategies as it cannot be automated and is necessary for each of them.

2. *Building a Language-Specific Meta-Model:* Similar to the previous step, defining the meta-model for Matlab/Simulink was considerably easier than for the different statechart dialects. On the one hand, the discussions with experts were a great help to build a meta-model conforming to their understanding of Matlab/Simulink. As such additional input was not available for the statechart dialects, we ended up with executing multiple iterations until we had built a meta-model providing all required details. On the other hand, we learned that building a meta-model for multiple dialects of a language is much more complex than we expected. In many cases, small not obvious differences between the analyzed dialects hinder an easy adaptation so that a systematic analysis and comparison of their concepts is crucial. However, despite the incremental realization of the statechart meta-model, we were able to successfully realize it for a corresponding Family Mining implementation. Furthermore, we found that such an incremental adaptation is supported by our proposed approach and even can ease the adaptation. Starting with the general structure of the language (e.g., the general nodes and edges) and increasing the complexity (e.g., adding their properties), it is possible to validate each edit step in the meta-model prior to adding further details. While defining a meta-model using our Vampire DSL requires some familiarization to its constructs, we found that one gets used to the notation quite fast. However, the true benefit becomes only apparent in the next steps of the adaptation as large portions of, otherwise manually, developed code (e.g., for the metric or the adaptation of the algorithms) can easily be generated. A solution for future work could be a graphical editor similar to the default EMF meta-model editor. This way developers could use the generation facilities of the Vampire DSL without even noticing that they apply an approach different to the meta-model editor, which they are familiar with.

3. *Defining a Custom-Tailored Metric:* We found that selecting appropriate weights for languages with large sets of properties is not a trivial task. For instance, statecharts comprise a much larger set of relevant properties compared to Matlab/Simulink models and, thus, ranking

Approach		Vampire DSL	Compare		Metric**	Match	Merge
			Generic	*Concrete***			
Custom	*MS*	–	–	213	475	347	1,271
Implementation	*SC*	–	–	310	1,017	347	1,520
Manual	*MS*	–	*324**	111	475	*347**	1,271
Adaptation	*SC*	–	*324**	143	1,017	*347**	1,520
Vampire DSL	*MS*	31	*324**	48***	–	*347**	1,271
Adaptation	*SC*	142	*324**	–	–	*347**	1,520

*reused from the generic framework **generated for the Vampire DSL Adaptation ***additionally implemented

Table 7.7.: *Lines of Code (LOC)* needed to adapt Family Mining for *MS = MATLAB/Simulink* and *SC = statecharts* using our guidelines and a Custom Implementation, a Manual or a Vampire DSL Adaptation.

them correctly is much more complex. As a result, we adjusted the weights multiple times before we found a solution that identified the expected variability relations (e.g., identified states with minor name changes as mandatory). A lesson learned here is that a clear structure for an implementation of the metric and its weights is crucial to keep an overview. Here, using the Vampire DSL is very helpful, because metrics can easily be generated with only a few additional lines of Vampire DSL description or corresponding EAnnotations in an existing meta-model. Thus, we favor this solution over the manual implementation of metrics.

4. *Using the Generic Variability Mining:* For this step, we found that using our framework and reusing the generic Family Mining implementation is a major help, because a Custom Implementation of the algorithms requires quite some effort. Executing either a Manual Adaptation or a Vampire DSL Adaptation largely reduces this effort as only small language-specific parts have to be manually implemented. Even more, it allows developers to focus their energy on implementing the algorithms to merge the identified variability in a 150% model representation and highlighting the details that are most relevant to them.

Overall, we can answer *RQ3.2a* positively as the adaptation of Family Mining for the selected languages is feasible and also an adaptation for multiple statechart dialects in a single realization is possible through the provided generic framework.

RQ3.2b Family Mining – Reduction of Implementation Effort To have a better understanding of the implementation effort involved in adapting Family Mining using the different strategies (i.e., Custom Implementation, Manual Adaptation and Vampire DSL Adaptation), we quantify this effort in form of LOC needed for the concrete adaptation. In Table 7.7, we present the corresponding numbers for the implementation of the *Compare*, *Match* and *Merge* phases as well as the *Metric* for the three strategies. Furthermore, we show the LOC needed for the Vampire DSL definition used during the Vampire DSL Adaptation. In case of the *Compare* implementation, we distinguish between a *Generic* part reused by implementations exploiting our generic Family Mining and a *Concrete* part as its extension for a specific language. Each row shows the LOC necessary to implement Family Mining for Matlab/Simulink *(MS)* and *statecharts (SC)* based on the three used strategies.

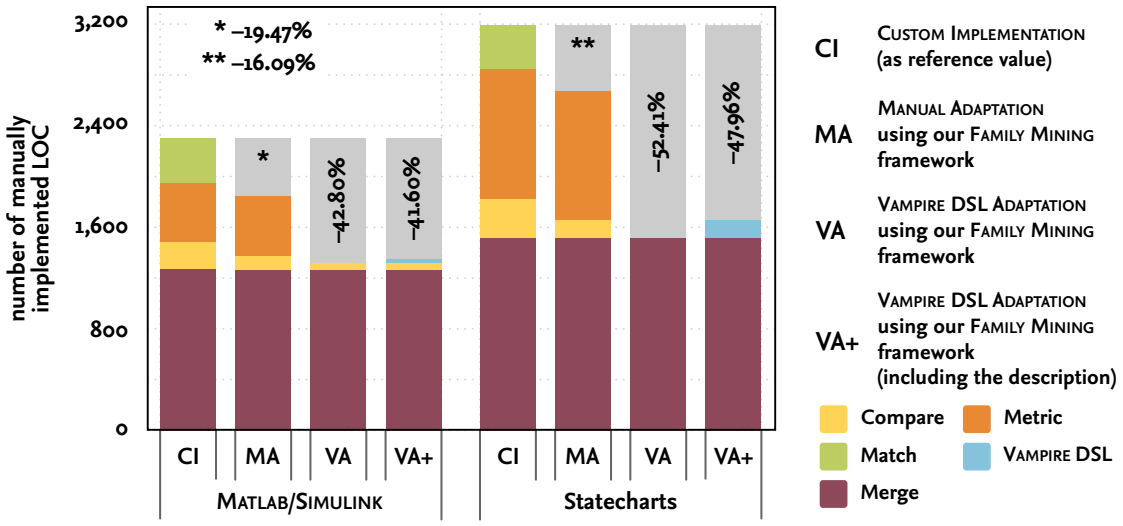


Figure 7.10.: Reduction of manually implemented LOC for the Manual Adaptation (MA) and the Vampire DSL Adaptation in comparison to a Custom Implementation (CI). The VA and VA+ distinguish between a reduction that neglects or counts the LOC for the Vampire DSL descriptions.

Looking at the presented numbers, we can see that the generic *Match* can be reused for the strategies based on our generic Family Mining. However, in case of the Custom Implementation it has to be manually implemented. Furthermore, the *Merge* algorithms are language-specific and have to be manually implemented for each adaptation of Family Mining. Also the *Metric* has to be implemented for the Custom Implementation and the Manual Adaptation as both only rely on the generic Family Mining pipeline and the provided data structures. In contrast, the *Metric* and the *Concrete Compare* algorithm adaptations can be generated for both languages during a Vampire DSL Adaptation based on the provided Vampire DSL descriptions. Only a small amount of manually implemented additional code is needed for Matlab/Simulink to allow selection of appropriate execution start nodes for the algorithms. Thus, the major difference exists in the effort for implementing the *Compare* and *Match* algorithms as well as the language-specific *Metric*.

In Figure 7.10, we summarize the reduction that can be achieved over the Custom Implementations of our Family Mining when using either a Manual Adaptation based on our generic framework or a Vampire DSL Adaptation. We show all manually implemented LOC necessary to adapt Family Mining applying one of the strategies and use the Custom Implementation (CI) as a reference value (i.e., 2,306 LOC for Matlab/Simulink and 3,194 LOC for statecharts). We show the reduction for an adaptation using the following strategies:

1. Manual Adaptation (MA).
2. Vampire DSL Adaptation *not counting* the lines that are necessary for writing the corresponding Vampire DSL description for the generation (i.e., VA).
3. Vampire DSL Adaptation *counting* the lines that are necessary for writing the corresponding Vampire DSL description for the generation (i.e., VA+).

For the analysis of the adaptation effort, we neglect the *Generic* parts and focus on adaptation-specific parts from Table 7.7 (i.e., *Concrete Compare* parts, the *Match* and *Merge* algorithms as well as *Metrics*).

The reduction is quantified in terms of LOC (cf. the *y*-axis) as well as the relative reduction. For example, when executing a Manual Adaptation for Matlab/Simulink based on our generic Family Mining framework, we have to manually implement 1,857 LOC. Compared to the Custom Implementation (i.e., 2,306 LOC) the reduction is 2,306−1,857 = 449 LOC (i.e., ≈ 19.47%).

Looking at the compared LOC for the different parts, we can see that using our generic framework for a Manual Adaptation of Family Mining for Matlab/Simulink models and statecharts can reduce the manually implemented LOC by about 18% on average (i.e., 19.47% and 16.09%, respectively). Using the Vampire DSL Adaptation strategy, we are even able to reduce the manually implemented LOC by about 48% on average for the two languages (i.e., 42.80% and 52.41%, respectively) compared to the Custom Implementation. Furthermore, even when considering the additionally needed Vampire DSL descriptions in these numbers, we can still achieve a large reduction of about 45% on average for both languages (i.e., 41.46% and 47.96%, respectively).

Of course, looking only at the quantitative reduction is not sufficient to give a realistic view on the needed effort. However, we can additionally summarize from our own experience that the provided generic facilities reduce the subjective adaptation effort over a completely Custom Implementation as only small language-specific parts (e.g., the selection of execution start nodes) have to be implemented in addition to the needed *Metric*. We further believe that the clearly defined interfaces for the implementation of language-specific parts support developers as the necessary extensions are made explicit through the implemented methods. Furthermore, applying the Vampire DSL reduces this effort even more, because for languages with clearly defined execution starts the Vampire DSL generator is able to derive all necessary code along with a language-specific metric. Thus, for such cases no manual effort other than writing the corresponding Vampire DSL description is needed to use our Family Mining for the adapted language. But even for languages where this additional selection is necessary, the effort reduction through the Vampire DSL is significant and we believe helpful to experts.

Altogether, we are confident that the provided tooling largely reduces the manual effort of adapting Family Mining for different languages, because a) the number of necessary LOC is significantly reduced and b) the necessary extensions are reduced to easy understandable and clearly defined interfaces. Although, we were not able to execute a user study in an industrial setting, we argue that both aspects lead to reasonable effort for adapting Family Mining for new languages. Thus, we overall answer *RQ3.2b* positively.

RQ3.3 Matador SPL – Language Independence For the realization of our Family Mining approach and corresponding tooling, we focus on providing variability mining solutions by allowing easy adaptation for different languages. Consequently, similar support for the Matador SPL approach is essential to allow a migration to an SPL independent of the used modeling language. Thus, we first analyzed whether the Matador SPL approach is capable of deriving the delta dialects necessary to generate the corresponding delta languages for the meta-models used by our case study subjects (i.e., the *BCS3* and *DAS2* model variants) and to encode the identified variability in a delta-oriented SPL. Based on the derived delta languages, we further generated the corresponding SPL artifacts storing the variability information identified by our Family Mining for the analyzed variants.

When analyzing the results, we found that the Matador SPL approach exhibits total precision and recall for the generated delta languages. Furthermore, we found that also the custom code to overcome DeltaEcore's missing support for initialization of `EAttribute` lists is correctly gen-

erated and that the corresponding custom delta operations are able to correctly use it. Comparing the delta languages derived by the Matador SPL approach with the delta languages generated by DeltaEcore, we were not surprised to find that the Matador SPL approach only generates a subset of all delta operations necessary for the meta-models of the corresponding modeling languages. As previously discussed (cf. Section 5.2), generating the subset of all possible delta operations has its advantages when developers want to limit the delta language for the used SPL to only necessary operations. For example, in cases where unwanted modifications of the created variants through specific operations should be limited. Thus, the subset of delta operations generated by the Matador SPL allows to migrate to an SPL *exactly* providing the delta operations necessary for encoding the identified variability. However, in case developers prefer to use a delta language providing all possible operations, it is still possible to use it for the SPL migration.

During further analysis of the results, we found that the Matador SPL approach exhibits total precision and recall for the generated delta modules. Furthermore, we found that these results are independent of the used approach to encode the variability (i.e., delta modules per feature or per complete variant) in the generated SPL and do not require any additional manual effort. Thus, migrating to an SPL using either approach is enabled by our Matador SPL approach in a language-independent way. However, users have to keep in mind that a migration using the delta module per variant approach requires additional manual effort to dissect the generated SPL into features and we recommend using available feature information to further automate this step.

Overall, we summarize that the Matador SPL approach is capable of deriving delta languages for the meta-models used by different modeling languages and delta modules using the corresponding delta operations. Thus, it can be executed for meta-models of different modeling languages without additional user input as all required artifacts can be generated from the provided meta-model and corresponding model variants. As a result, we argue that migration to an SPL using our Matador SPL approach is possible in a language-independent way and overall answer *RQ3.3* positively.

7.4.4. Results and Discussion of RQ4: Usefulness of Results

To evaluate the usefulness of the results generated by our approaches, we evaluate whether they support developers in the desired way and, thus, provide an actual benefit.

RQ4.1 Corevid – Improvement of Results To examine the results of the Family Mining approach with or without cluster and outlier detection provided by the Corevid approach, we distinguish two situations: 1) cases *with* outliers and 2) cases *without* outliers.

In case outliers exist, we identified that our outlier detection improves the fine-grained variability information generated by our Family Mining algorithm. The main reason is that outlier variants represent models that have a *low* or at worst *no* relationship to the remaining input models. Executing our Family Mining approach without detecting these outliers might result in unexpected variability relations in the 150% model or even elements that have no relation to the rest of the 150% model. As a consequence, developers might be unable to understand the variability relations identified by the Family Mining approach as information is presented that does not conform with their expectations. Examples are situations where developers left the company and knowledge about existing relations are lost or models from an acquired company are integrated in the common repository. Thus, we argue that using the outlier detection is essential in situations where users are not fully familiar with the input models and relations between these models are unclear.

In case *no* outliers exist, we do not necessarily need to execute our Corevid approach for a cluster detection as the generated results for the complete set of input models represent valid variability information. As a result, such design decision are up to the domain expert and highly depend on the selected use case. Executing Family Mining on particular clusters might allow users to focus their analysis on the corresponding models. Furthermore, detecting clusters reduces the chance of unexpected variability information (e.g., induced by variants from other clusters). For example, variants comprising a certain feature might result in unexpected Family Mining results for developers that are unaware of this feature (e.g., when different departments work in different projects on the same models). Thus, detecting clusters prior to executing the fine-grained variability mining can improve the experience of users. However, we think that these clusters should at least be evaluated at a high level by users as, depending on the focus of users, it might be interesting to combine multiple clusters for a bigger picture of the overall system. For example, this could be interesting for senior developers working on multiple projects and, thus, requiring a global overview over models that were identified as separate clusters.

Overall, we answer *RQ4.1* positively as using the cluster and outlier detection allows to improve the experience of users. The generated 150% models are tailored towards their expectations as outliers are removed and the identified variability information can be focused on particular clusters.

RQ4.2 Family Mining – Usefulness of Results To evaluate the usefulness of the results provided by our Family Mining approach, we executed expert interviews with engineers from the industry partner that provided us with the *Industrial* models. The corresponding detailed results can be found in Section 7.5 and we leave their discussion and the answer for *RQ4.2* to this section.

RQ4.3 Matador SPL – Usefulness of Results For the evaluation of our Matador SPL approach, we were not able to perform user studies for migrating existing model variants to an SPL. Main reason are the costs for companies that are linked with letting engineers participate in such experiments.

Nevertheless, due to the positive results regarding the correctness of the SPL migration (cf. the discussion of the results for *RQ1.3a* and *RQ1.3b* in Section 7.4.1) and the corresponding low runtimes (cf. the discussion of the results for *RQ2.3* in Section 7.4.2), we argue that our Matador SPL approach actually helps experts to automatically migrate existing variants to an SPL with low effort. Furthermore we showed in Chapter 5 that it is possible to automatically generate feature models (cf. Figure 5.2) with corresponding delta modules implementing the features from existing model variants. In addition, it is possible to apply our provided refactoring operators to restructure the feature model (cf. Figure 5.4) to a representation similar to the ground truth of an existing SPL (cf. Figure 2.9). As we were able to confirm these results during our evaluation for all SPLs generated from the *BCS3* and *DAS2* model variants, we are confident that our claim holds.

Furthermore, the possibility to refactor the generated SPL towards a solution that is similar to an SPL created according to SPL best practices (cf. Section 5.5) shows that combining the completely automatic migration to an initial SPL with manual restructuring using domain knowledge is a valid approach. As a matter of a fact, we argue that this is the only sensible approach to perform the migration of variants to an SPL, because automatic algorithms will never be able to generate implementations that are structured according to the expectations of experts. As a result, the Matador SPL approach is able to minimize effort for encoding the variants variability in an SPL realization, but leaves full control over the final result. Altogether, we answer *RQ4.3* positively.

Participant	Fields of Expertise	Role in the Company
P1, P6	• Model-Based Development • Static Analysis	Senior Developer, Architect
P2 – P5	• Model-Based Development • Static Analysis	Senior Developer
P7	• Model-Based Development • Static Analysis	Technical Leader
P8	• Model-Based Development • Static Analysis • Research	Senior Developer

Table 7.8.: Participants of the executed expert interviews.

7.5. Expert Interviews

Evaluating only objective aspects of our approaches (i.e., their runtime, scalability and correctness) is not enough to evaluate their applicability in an industrial setting. In addition, it is important to get feedback from developers in the potential target group of the developed techniques. Thus, we executed a *semi-structured* [Pat87] interview with engineers from the industry partner that provided us with the *Industrial* Matlab/Simulink models. The overall goal was to answer *RQ4.2* by getting an understanding of the experts' expectations regarding variability mining and to what extent our Family Mining meets them. Furthermore, we were interested in the fields of application for the proposed approach and, most importantly, the usability of the identified variability information. The executed study is based on *grounded theory* [GS67] a systematic approach from social sciences that is increasingly used in software engineering. The approach systematically encodes and categorizes expressed subjective views (e.g., from interviews) to link them together in theories on the analyzed topic (e.g., our Family Mining) and check whether they hold up through all interviews.

7.5.1. Interview Participants

The eight participants of our interviews have between 10 and 30 years of experience in developing Matlab/Simulink models and their maintenance. At the time of the interviews, all of the participants were working in the area of model-based development of Matlab/Simulink models and their static analysis. In addition, one of the participants was involved in research on model-based development and more specifically on developing Matlab/Simulink models. In Table 7.8, we show a summary of our participants in the expert interviews.

7.5.2. Data Collection

All interviews were performed in separate sessions with the participants not knowing the questions prior to the interview. In addition, all answers given by the participants were not accessible by other participants to prevent them from being biased. Although all participants were familiar with the general idea and approach of our Family Mining, a short presentation on the applied techniques was given prior to each interview to refresh the participants' memories.

Questionnaire Our questionnaire for the interviews comprised nine questions targeting three areas most interesting to us. In Appendix G, we show the complete questionnaire with short descriptions on the goals of each question. All questions were formulated to be open-ended using the guidelines from [CW03]. In the first part, *General Expectations*, we asked the experts about their expectations regarding the Family Mining approach and its results. In the second part, *Fields of Application*, we were interested in the experts' opinion on the possible application scenarios and, for example, whether applying our approach for a migration towards an SPL realization would be of use to them. Furthermore, we asked whether the experts prefer high precisions and possibly longer runtimes or whether they would rather sacrifice some precision of the approach to reduce the overall runtime. In the third part, *Usability of Results*, we asked for the experts' opinion on the usefulness and quality of the Family Mining results and if there is room for possible improvement. The answers to these questions in the three areas of interest allow us to answer the last open research question *RQ4.2* from Section 7.4.4, because we get a clear understanding of the users expectations towards the approach, their foreseen field of application and their assessment of the usability.

Interview The interviews were performed in sessions of 25 to 35 minutes. Before starting the interview, two models from the *SPES_XT* case study were provided and the interviewee was asked to identify the variability. This way, we were able to gather unbiased information on the experts' perception of the variability between the models. For the automatic identification of the variability relation between the example models, we used the Matching Window Analysis (MWA) algorithm (cf. Section 4.7). Based on the results, we asked the experts to what extent the automatically generated results of our approach captures their previously manually identified variability.

7.5.3. Interview Results

In this section, we summarize the results from the expert interviews and discuss them at the end of this section with respect to research question *RQ4.2*.

General Expectations All (100%) of the participants stated that they see support of their work during model development and maintenance as the key purpose of our Family Mining. As main reason for the need of variability mining techniques, such as our Family Mining, the participants *P1*, *P2*, *P7* and *P8* (50% of the participants) mentioned lacking documentation during model evolution. According to these experts, this lacking information results in risks for the maintainability of the developed models and additional effort necessary to recover or validate performed changes. All (100%) of the participants expressed that they do not demand a flawless but sufficiently precise analysis. The rational behind this is that all (100%) of the participants expect manual inspection of the results generated by the Family Mining approach to be necessary until enough trust is built. The participants *P1*, *P2* and *P4* (37.5% of the participants) mentioned that identifying both common and varying parts in analyzed models as key to their work. Furthermore, they specifically highlighted the Family Mining's ability to combine both information as an advantage over common clone detection and differencing tools.

Fields of Application The opinions on the possible use cases differ amongst the interviewed experts. Participants *P2*, *P4* and *P7* (37.5% of the participants) categorize our Family Mining approach as valuable in a reverse-engineering scenario. In addition, *P4* and *P7* (25% of the participants) specifically highlighted the Family Mining's ability to reveal reusable parts in existing models when

developing new systems from scratch. However, *P3* mentioned that refactoring scenarios are not imminent in their company at the current point of time and explained that variability is rather considered during the design phase of developed systems instead. This assessment is backed by *P5* as currently realization of variability is approached in a constructive manner rather than using reverse-engineering. Furthermore, *P8* explicitly mentions the Matlab/Simulink constructs discussed in [WM13] to be the means of choice for explicitly modeling the variability in their models. In addition, the participants *P1* to *P5* (62.5% of the participants) currently see the primary use case for our Family Mining approach to revolve around analyzing two models and *P4* mentions the importance of change traceability in large model repositories. However, all (100%) of the participants foresee that the capabilities provided by our Family Mining approach will gain more importance for their company in the upcoming years. Main reason is that development of models for new products will increase the need for efficiently identifying reuse potential in existing variants.

Usability of Results When asked about whether or not the family model reflects their initial perception of variability, all (100%) of the participants, without exception agreed with the contained information. Furthermore, *P1*, *P3* and *P5* (37.5% of the participants) specifically highlighted our explicit categorization and annotation of identified variability relations into mandatory, alternative and optional parts as valuable for their work in model maintenance. The participants stated that these captured details are crucial and, thus, well appreciated to support developers in getting an initial impression of model relations prior to executing their current task. Although the *ELF* and *DTM* models were partitioned for processing, *P6*, *P7* and *P8* (37.5% of the participants) do not see this as a drawback but specifically mention the users' domain knowledge allowing for a precise targeting of certain model parts for comparison and aggregation of the results. During our evaluation, we found that few hierarchy shifts and horizontal dispersions were identified (cf. Section 7.4). These findings were confirmed by all (100%) of the participants. Furthermore, *P1* and *P2* (25% of the participants) explained that the current practice in their company is to develop a parameterizable 150% model of the functionality in all variants. This also explained the few identified differences in the analyzed models. *P7* and *P8* (25% of the participants) further added that these horizontal dispersions and hierarchy shifts are normally not present when following the company's development guidelines and are only introduced in critical situations. For example, when applying hotfixes for bugs that were identified prior to a major release, developers might introduce these horizontal dispersions and hierarchy shifts as a temporary solution. However, after the final release, these violations to the company's guidelines are normally rectified by the developers. As a result, *P7* and *P8* (25% of the participants) highlighted that information provided by our Family Mining on corresponding occurrences is indeed appreciated by executing developers to identify such scenarios more easily.

Overall, the interviewed domain experts stated that they generally appreciate our approach and the resulting variability information is considered understandable. Furthermore, the customizable metric and the possibility to introduce additional techniques to the framework are welcomed as they allow easy customization of the algorithms. In future work, the experts would like to see a direct integration of the developed algorithms within Matlab/Simulink. This way, they would not have to switch between their development environment in Matlab/Simulink and Eclipse for applying our Family Mining approach.

Although, the experts wish for a tighter integration with their main development tools, they are overall positive about the presented approach and its support for their work. Furthermore, they

agree with the identified variability information and see potential to use such details in their everyday work. Thus, we answer the remaining research question *RQ4.2* from Section 7.4.4 positively.

7.6. Threats to Validity

Following the guidelines by Runeson et al. [RH09], we identify the following threats to validity that are inherently present in our evaluation, although, we designed, implemented and evaluated our approaches with great care.

7.6.1. Construct Validity

During the executed evaluation, we analyzed the scalability of Family Mining approach for different case study subjects. While other measures might exist to evaluate the scalability, we think that measuring the runtime and setting it in relation to the sizes of the analyzed models shows the scalability for growing model sizes. Furthermore, our corresponding results were backed by the analysis of created comparison elements in relation to the model sizes. Thus, we are confident that the selected measures show representative results for our Family Mining approach.

During further analysis, we evaluated whether the effort of adapting our Family Mining for new languages is reduced by its generic implementation and the provided Vampire DSL. To this end, we measured the LOC necessary to realize such an adaptation using a completely Custom Implementation of Family Mining, a Manual Adaptation relying on the generic Family Mining and a Vampire DSL Adaptation. Using the LOC as measure, we only investigate the quantitative effort of adapting Family Mining for new languages and we ignore other influences, such as human factors or understandability of the general approach. As we did not execute a user study to investigate these factors, this threat remains. Nevertheless, we are confident that beside reducing the necessary LOC to realize Family Mining, our provided tooling actually reduces the perceived effort. Most importantly, the necessary manual extensions are reduced to a minimum (i.e., a small number of clearly defined interfaces) and developers can rely on extensive adaptation guidelines for analyzing new languages and defining corresponding adaptations (cf. Chapter 4).

7.6.2. Internal Validity

Different aspects influence the results of our evaluation, and we might not have considered all of them. For our runtime and scalability analysis, we rely on measures in a non-closed system and, thus, we cannot rule out the possibility of other factors influencing the execution on these systems (e.g., through scheduling of the operating system between different tasks). However, to reduce the influence of such factors, all executions to measure runtimes were performed 10 times and the average was calculated. While the threat remains, we are confident that our countermeasures reduce the effects of such factors and allow generalizability of the results to some extent. Thus, the presented runtimes and scalability analysis should at least give an indication of the approaches' behavior under realistic circumstances.

In addition, our similarity metric for the Family Mining approach uses weights and thresholds that might be hard to specify upfront and that may influence the results. Such weights and thresholds represent heuristics and are highly dependent on human intuition and experience of the implementing developer. Consequently, results generated based on metrics might not always conform with the results expected by other developers. As we might not have considered all factors

for the created metrics and, thus, unexpected behavior is possible for different scenarios, this is a threat to validity. However, we have created our metrics with caution and only after carefully analyzing possible dependencies between the used weights for different properties of model elements. In addition, the correctness of the results was evaluated by three different developers and eight experts from our industry partner independently confirmed that such results conform to their expectations and are useful to them. As a result, we are confident that the results should be at least close to the intuition of other experts. Furthermore, we allow for easy adjustment of metrics through the GUI of our Family Mining framework and, thus, experts are able to specify these weights conforming to their own intuition.

The Corevid approach relies on different IR and NLP techniques to identify clusters and outliers from approximate data derived from analyzed models. As a result, the approach is subject to a variety of factors, such as the user-specified extraction scheme or applied weighting, that might influence its results. While we were able to show the approach's general capabilities of identifying clusters and outliers, this threat remains as the results for other models might differ. Nevertheless, we are confident that the Corevid approach will show similar results for other cases as the Samos framework has been successfully evaluated with different large and realistic data sets before (cf. [Bab16, BCB16, BCV+16, BC17, BCB18]).

7.6.3. External Validity

During the evaluation of our approaches, we evaluated adaptability of our generic Family Mining approach for different scenarios and block-based languages. This evaluation is subject to external threats to validity as it is executed for two different modeling languages only. Thus, the generalizability of the adaptability for our Family Mining approach is limited as other languages might comprise additional characteristics not supported by our approach. While we selected the considered languages and case study subjects with the goal to evaluate a variety of potential scenarios, this threat remains. Nevertheless, we are confident that the selected languages are representative for modern modeling languages as they use different well-known paradigms (i.e., dataflow-oriented vs. state-based execution) with a variety of different concepts. For instance, statecharts allow to express parallel execution and provide transitions with a concrete behavior defining their identity. In contrast, Matlab/Simulink models completely rely on hierarchical decomposition and only provide limited information through their connectors.

For the evaluation, we also investigated whether the effort of adapting our Family Mining approach for new languages is reduced through the used generic implementation and the provided Vampire DSL. During this evaluation, we focused on a quantitative analysis by means of LOC needed to realize the necessary adaptations and presented the subjective view of a single developer without performing a corresponding user study. As a result, this evaluation is subject to external threats to validity as the findings cannot be generalized. Nevertheless, we are confident that our provided tooling is helpful to other developers and actually lowers the barrier of adapting Family Mining compared to a manual reimplementation of all algorithms. For example, the provided tooling through the Vampire DSL is able to reuse existing meta-models and to generate large portions of the necessary adaptations using descriptions that are close to standard meta-modeling languages. Furthermore, the limited number of methods that have to be implemented in clearly defined interfaces additionally reduce the effort for developers executing the adaptations.

During the evaluation, we investigated correctness and usefulness of variability information identified by our Family Mining EFA and MWA algorithms. This evaluation is subject to external threats to validity as:

1. The evaluation of the correctness was performed by three developers. Furthermore, our expert interviews were performed in a single company focusing on the development of software for automotive applications. Other researchers or developers might question the correctness and usefulness of the identified variability as their expectations regarding these relations might differ.

2. The evaluation was limited to three case studies from the automotive domain with two different languages. Our generic Family Mining might not be able to identify correct variability for other modeling languages or models.

3. The selected case study subjects are limited to a single domain only.

We acknowledge that other domains may exhibit peculiarities, which were not considered by us and which may adversely affect our techniques. However, we implemented our generic Family Mining without having a particular domain in mind and prior to our evaluation. Thus, we kept ourselves from being biased and, with the automotive systems exhibiting a relatively high complexity, we argue that our results are representative to some extent for identifying realistic variability relations. In addition, given the interviewees' experience and their mostly coinciding and positive responses, we argue that the executed expert interview supports this assessment as it provides us with at least a strong indication that our technique is appreciated and valuable to model engineers. Furthermore, we performed a direct and extensive manual evaluation of about 110 model comparisons and an indirect evaluation of 15 additional model comparisons (i.e., through the results of the Matador SPL approach). Thus, we are overall confident that the Family Mining approach is capable of identifying variability information for different case studies and other domains as well, given the complexity of the used subjects (e.g., the used *Industrial* subjects) and the fact that all case studies are at least close to or even represent real-world productive systems.

Another threat to validity is the limited size of the used model variants of the *BCS* case study as well as the fact that these models are generated from an SPL and, thus, might not represent realistic clone-and-own use cases. However, we think using the *BCS SPL* for our evaluation shows that our approach can identify variability conforming to SPLs that were developed using corresponding best practices. Besides, this provides us with a ground truth and we do not have to manually identify variability information that might be raised to question. In addition, the *BCS SPL* also contains minor differences between the implementations of features (e.g., to realize proper interaction with additionally selected features). Thus, our approach shows that it is capable of identifying differences that might be regarded as accidental differences introduced in clone-and-own scenarios.

For the evaluation of the Family Mining approach, we use meta-models for Matlab/Simulink and seven statechart notations. While our evaluation showed that the meta-model for Matlab/Simulink is expressive enough to support corresponding Family Mining, we only evaluated the statechart meta-model for the IBM Rational Rhapsody notation. Furthermore, the considered *BCS* subjects do not comprise all language elements supported by IBM Rational Rhapsody (e.g., final states). Nevertheless, we are confident that the meta-model is also capable of handling the other

notations as we executed a detailed analysis of all mentioned dialects in [Wil14]. For this analysis, we followed an approach that served as basis for the guidelines in Section 4.1 and Section 4.2 by carefully selecting and evaluating relevant documents and constructing the corresponding meta-model only afterwards. While the threat remains, we are confident that the created meta-model should be able to handle the considered notations due this careful design. Furthermore, as we designed and implemented all language-specific parts of the Family Mining, such as metrics, with keeping language elements from the other notations in mind, our algorithms should be capable of handling these elements.

The evaluation of our approaches' runtimes and scalability is subject to external threats to validity as the results might differ for other case studies and languages. However, we designed the presented algorithms without having a particular domain in mind and prior the evaluation using the selected case studies. Thus, we kept ourselves from being biased and are confident that the performance of the algorithms is at least similar for other scenarios.

For the evaluation of the Corevid approach, two developers focused on a single case study without involving other users. Thus, the generalizability and adaptability of the results is limited. However, as finding realistic and sufficiently large case studies with a corresponding ground truth is hard, the threat remains. Nevertheless, we are confident that the results are representative to some extent as the underlying Samos framework was successfully evaluated with similar (meta-)model clustering scenarios and different languages before (cf. [Bab16, BCB16, BCV+16, BC17, BCB18]).

Furthermore, we did not execute a detailed scalability analysis of our Corevid approach due to the limited size of the considered case study. Nevertheless, we are confident that the approach will scale for other even larger scenarios as it was previously evaluated for scenarios with up to about 7,300 models [BCB18]. Furthermore, as previously discussed for the results of *RQ2.1*, we argue that the runtimes of multiple hours for such larger scenarios are within a reasonable time frame, because the approach will not be executed on a daily basis.

Another point, already addressed in the discussion of the results of *RQ1.1a* and *RQ1.1b*, is that the ground truth for the evaluation of the Corevid approach is created by selecting or deselecting features for the generated variants. Hence, the expectation of the domain experts is shaped with respect to the high level features of the *BCS SPL*. This contrasts with the clustering technique working on the level of implementation (i.e., the model variants). Elaborate weighting schemes based on features and/or model elements could be introduced to mitigate this situation, where the domain experts associate the corresponding parts with varying importance to guide the clustering. However, given the focus of the thesis, we leave definition and application of such weighting schemes for future work. While the threat remains, we are confident that our results are representative for the capabilities of our Corevid approach, because it showed for the evaluated models the expected results. Furthermore, it was successfully applied in other scenarios with differing requirements (cf. [Bab16, BCB16, BCV+16, BC17, BCB18]).

For the evaluation of our Matador SPL approach's correctness and language-independence, a single developer analyzed the results for only two case studies without executing a detailed user study. This limits the generalizability of the results, because other case studies using different meta-models might reveal scenarios that were not considered by us. Nevertheless, we are confident that the approach is capable of handling other languages and case studies as we designed all algorithms independent of the selected case studies or a specific language in mind. Furthermore, we selected

models from two case studies with different modeling paradigms (i.e., dataflow-oriented for MATLAB/Simulink vs. state-oriented for statecharts) and meta-models of different complexity to be confident that our delta generation approach is applicable for a wider range of models. In addition, the only language-specific part is the feature identification as it relies on the hierarchical decomposition of developed models using corresponding language constructs. The general Matador SPL approach is realized in a language independent way, completely relying on the Ecore meta-model and language-independent variability information from the analyzed 150% model. Thus, we are confident that Matador SPL is capable of deriving delta dialects and delta modules independent of the used meta-model.

For the evaluation of our Matador SPL approach, we did not execute a detailed scalability analysis, which limits the generalizability of the approach. Similar to the evaluation of the Corevid approach, finding realistic and sufficiently large case studies with a corresponding ground truth is hard. Although we had access to the *Industrial* case study models, we were not able to execute scalability evaluations for our Matador SPL approach using these models as the corresponding time frame was limited. While the threat to validity remains, we are confident that our approach scales for larger and even more complex models as it showed reasonable performance for the evaluated cases. Furthermore, the approach realizes a translation of existing variability information to a different representation without involving any complex comparisons or calculations. Thus, we see the scalability of the Family Mining approach as the limiting factor, because the Matador SPL approach relies on the corresponding details. As a result, we are confident that the scalability for larger models is given as the Family Mining approach showed to scale and identify variability relations for industrial models within seconds. Furthermore, even runtimes of hours for the Matador SPL approach would be acceptable as the migration to an SPL is not executed on a daily basis.

Another threat to validity is that both meta-models used in the cases studies were created by authors of the thesis. Thus, a similar modeling style might be used and generalizability of the results could be raised to question. However, as the meta-models were created only after detailed analysis of the languages and according to best practices in model-driven development, we are confident that their design is close to solutions of other developers using similar guidelines.

7.6.4. Reliability

The results of our Family Mining approach rely on heuristic metrics and thresholds. As a result, the identified variability information might not be generally reliable, because different factors (e.g., other requirements or models) might lead to incorrect results. Thus, also the Matador SPL approach is subject to this threat to validity as it builds upon the Family Mining results. While the threat remains, we are confident that the general design of our algorithms and the ability to adjust all thresholds and metrics (or even build own ones) allows to easily account for this. Furthermore, our evaluation showed that our Family Mining approach is able to provide results that are evaluated by different developers to be correct and even convince industrial experts of the usefulness.

The Corevid approach aims to deliver a fast but approximate overview of the given data. Hence, it may not be ideal if very high accuracy is required, and the approach might not be regarded as reliable in every situation. While the threat remains, this is an inherent assumption of the algorithm as it focuses on identifying clusters and outliers for large data sets. To achieve corresponding results in reasonable time, the algorithm relies on abstracted data and, thus, only provides potentially

unreliable data. Nevertheless, the general algorithm showed that it is able to provide the expected results as it was evaluated with different large and realistic data sets (cf. [Bab16, BCB16, BCV+16, BC17, BCB18]) and also showed the expected results for our case study subjects. Furthermore, developers are easily able to adjust the analyzed data and the applied settings through the highly configurable Samos framework. As a result, we are confident that the Corevid approach is able to provide the expected results in different situations.

7.7. Chapter Summary

In this chapter, we have evaluated the techniques proposed in Part II of this thesis. To this end, we applied them to three large-scale case studies with industrial background and analyzed them with respect to *correctness and precision*, *runtime and scalability*, *adaptability* and *usefulness* of the results.

During our evaluation of the Corevid approach, we were able to show that the approach is able to identify sensible clusters and outliers in large sets of variants with reasonable runtimes. Due to the design of the underlying Samos framework by Babur et al. [Bab16, BCB16, BCV+16, BC17, BCB18], the approach is easily adaptable to other modeling languages by replacing the used extraction schemes. Based on the identified clusters and outliers, developers can easily remove unrelated variants from the input sets and analyze different clusters.

During our evaluation of the Family Mining approach, we were able to show that the approach is able to identify correct and precise fine-grained variability information between input variants. The approach exhibits reasonable runtimes and scales even for large models. Due to the provided generic Family Mining framework and corresponding tooling (e.g., the Vampire DSL), developers can easily adapt the proposed approach for different languages with low manual effort. When interviewing domain experts from our industry partner, we have got overall positive feedback regarding the quality and usefulness of the variability information provided by the Family Mining approach.

During our evaluation of the Matador SPL approach, we were able to show that the approach is able to correctly dissect existing model variants based on 150% models created by the Family Mining approach into sensible feature implementations. Using corresponding SPL realizations, we showed that it is possible to automatically generate all input variants as well as additional variants enabled by the new SPL. As the generic SPL generation algorithm fully relies on the Ecore meta-model structure of analyzed variants and the possibility to generate necessary delta languages for the applied modeling languages, our Matador SPL approach is easily adaptable for different settings. By applying the provided refactoring operators, the generated SPL realization can easily be refactored while preserving its functionality and, thus, the Matador SPL approach enables migration of model variants to a custom-tailored SPL.

Overall, we showed that all techniques exhibit reliable and useful results that can easily be generated for different modeling languages even in scenarios with large-scale models. Thus, the techniques proposed in this thesis are able to semi-automatically migrate a set of existing model variants with possibly unclear relations to a software product line, while making low-level variability relations explicit to experts.

8 Conclusion

This chapter concludes the thesis with a summary of our contributions (cf. Section 8.1). Next, we discuss design decisions that were made for the described approaches (cf. Section 8.2). Finally, we present possible future application areas of the approaches (cf. Section 8.3).

8.1. Contribution

In this thesis, we successfully showed how sets of existing model variants with potentially unclear relations can be migrated to a corresponding SPL realization. In the following paragraphs, we summarize our results with respect to our three research questions (cf. Section 1.3):

Research Question RQ1 – Identifying High-Level Variability Relations To identify high-level relations between model variants whose relationships are not known by users, we presented our Corevid approach (cf. Chapter 3). The approach takes as input a set of model variants conforming to a language-specific meta-model and uses a corresponding user-provided extraction scheme to extract and analyze relevant information for the model variants. By applying a user-selected distance measure the approach is able to generate dendrograms visualizing the identified high-level similarity of compared model variants. Using these dendrograms, users can easily identify clusters of highly related variants and outliers with low or even no similarity to the remaining variants. During our evaluation, the approach showed for different data sets that it is capable of identifying expected relations for such scenarios.

Research Question RQ2 – Identifying Low-Level Variability Relations To identify low-level relations between related model variants (e.g., created by using a clone-and-own approach), we presented our generic Family Mining approach (cf. Chapter 4). The approach exploits the common structure of block-based modeling languages to perform detailed analysis of model variants by executing the concrete comparisons using a user-adjustable metric and decision wizards to support the matching. As a result, the approach can easily be adapted to new languages by exchanging the underlying meta-model and defining new metrics. Both steps are eased by our guidelines and our Vampire DSL to semi-automatically generate the necessary meta-model, metrics and large parts of the required adaptations for our Family Mining algorithms. Overall, the Family Mining approach iteratively compares the input model variants and merges the results into annotated 150% models storing all model elements of the variants with their identified low-level variability relations. Based on this variability information, developers can easily perform detailed analyses of software families and improve their maintenance (e.g., by directly seeing which variants comprise a specific bug). During our evaluation, the approach showed for different case studies with industrial background that it identifies the expected variability relations in reasonable runtimes and scales for industrial models.

Research Question RQ3 – Migrating Existing Variants to an SPL To migrate analyzed model variants based on their identified low-level relations to a delta-oriented SPL realization, we presented our generic Matador SPL approach (cf. Chapter 5). The approach is able to automatically derive a delta

language with delta operations specific to the used modeling language and, thus, allows SPL migration for languages without existing variability modeling support. By encoding the identified variability information in delta modules conforming to such delta languages, the approach enables an easy and fully automatic migration of model variants to an SPL. Furthermore, users can support the Matador SPL approach by providing additional details on features of input variants. Based on this additional information, it is possible to dissect the existing functionality into delta modules that later can be used to derive different variants. During our evaluation, the approach showed that it can successfully dissect implementations of existing variants into reusable SPL artifacts and, thus, enables their migration to SPLs allowing not only derivation of these input variants.

Overall Summary In Figure 8.1, we show a high-level overview of the proposed solution explicitly highlighting the artifacts that have to be provided by the user to execute *Custom-Tailored Product Line Extraction*. As we can see, only a small number of artifacts has to be provided by the user to allow automatic analysis of variability for model variants and their migration to an SPL. Furthermore, our provided guidelines and corresponding tooling largely reduce the linked effort as most artifacts can be generated or only require small manual extensions in clearly defined interfaces.

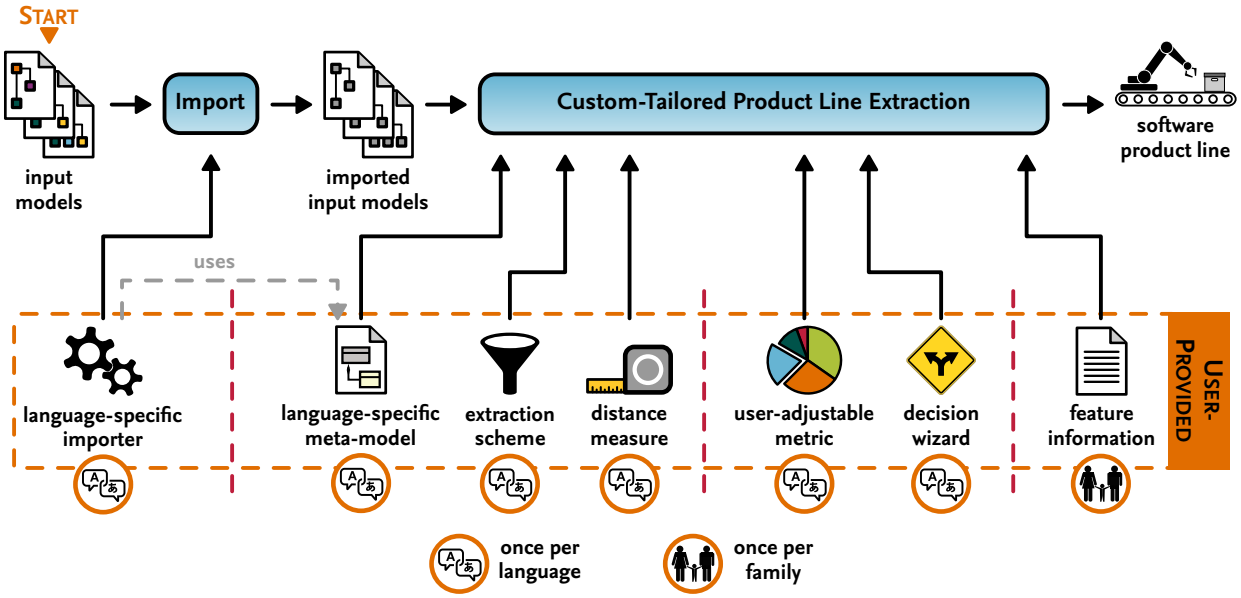


Figure 8.1.: High-level view of the overall *Custom-Tailored Product Line Extraction* approach.

Overall, the presented techniques are able to advance the state of the art. First of all, the Family Mining approach is able to make fine-grained variability relations explicit to developers by applying generic algorithms relying on language-specific metrics. This way, applying variability mining is not limited to specific languages, but can easily be adapted for a multitude of languages and different scenarios. Based on these results, migrating existing product variants to a corresponding SPL realization can be easily achieved by applying our generic Matador SPL approach. However, most importantly, our Corevid approach is the first to support developers in understanding coarse-grained relations between model variants prior to a detailed analysis of their variability and migration to an SPL. Thus, in contrast to the state of the art, our approach allows to identify sensible clusters of related variants and to remove unrelated outliers, which leads to an improved overall

quality of the generated SPL in scenarios where such relations were lost. Altogether, the presented solution is able to semi-automatically migrate a set of existing model variants with possibly unclear relations to a software product line, while making low-level variability relations explicit to experts.

8.2. Discussion

During the design of the approach described within this thesis, we made certain design decisions that influence the behavior and applicability of the developed algorithms.

Selection of Metric Weights and Thresholds Our Family Mining approach relies on heuristic comparison of model variants using metrics to calculate the similarity of model elements. Furthermore, we employ thresholds to identify the variability relations between model elements. As we have discussed in Chapter 4, developers can easily adjust the corresponding values for their current use case or the used language. One question that comes to mind when configuring these metrics and thresholds is whether there is the "perfect" parameter set for the particular use case and how we can find it? We agree that there is no universally valid solution to this question because expected results highly depend on the users experience and expectations. While many developers agree with the metric weights and thresholds selected in this thesis, others might disagree and expect slightly different similarity values for compared model elements. Thus, finding the correct parameters for the current use case might involve some adjustments until stakeholders agree with the shown results. However, to ease this step and allow full flexibility, we designed the corresponding parts of the Family Mining to be highly adjustable. Furthermore, we provide detailed guidelines and discuss our experiences on possible approaches to find weights and thresholds in Chapter 4.

Input Order of Analyzed Model Variants One question that came up quite a few times when discussing our Family Mining approach with colleagues and reviewers is whether the input order of the compared model variants has an influence on the results? While we did not execute a formal proof, we performed a variety of tests with different sets of input models from our evaluation subjects. During these tests, we did not find any differences between the results for varying input orders. We argue that this is mostly due to the iterative merging of results into a single 150% model. This way, the Family Mining always considers model elements from previous comparisons and stores all relevant information of elements in the merged 150% model to not loose any details. As a result, there should not be any differences between executions with varying input orders.

Generation of Delta Dialect with Subset of Delta Operations We already discussed in Section 5.2 that the delta dialect derived by our Matador SPL approach might only represent a subset of all possible delta operations for a used meta-model. Driving factor behind this design decision is to limit the number of available delta operations to not overwhelm developers and to constraint possibly unwanted modifications of the implementation artifacts. This way, we provide developers only with the delta operations that are actually necessary to realize an SPL for the input variants. In general it is possible to extend the generated delta dialect with further delta operations. However, we agree with Schuster et al. [SSS17] who argue that during medium-term development of an SPL (i.e., with an already large but still growing SPL) the management of growing variability can become confusing. As a result, Schuster et al. [SSS17] propose to transition to an SPL realization using so-called *variability interfaces* to explicitly show which parts of the SPL implementation can be changed. This allows to hide unmodifiable and irremovable parts of the SPL and developers can focus on the

developed variability. Furthermore, developers explicitly know which parts of the implementation stay the same and can assume them to be unchanged in all developed variants. Thus, generating only the necessary subset of delta operations aims in a similar direct while still providing the full flexibility of extending the delta dialect with further operations.

Delta-Oriented Solution vs. Clone-and-Own Approach For migrating a set of existing model variants to a corresponding SPL realization, we decided to use delta modeling. Main reasons are its high flexibility (i.e., developers can easily extend the created SPL) and the easily understandable language constructs due to the delta operations' proximity to operations in the original notation. However, another factor that should not be neglected is that delta-oriented SPLs and clone-and-own strategies require similar operations to create new variants. The initial cloning step is executed by copying a variant in case of clone-and-own and by selecting an existing variant in case of delta-oriented SPLs. The editing step is performed in both cases by adding, removing and modifying functionality. However, while in case of clone-and-own these modifications are *directly* executed on the copied variant, they are only performed *indirectly* through the created delta modules and corresponding delta operation calls in case of delta modeling. Overall, we think that this additionally supports developers in transitioning from clone-and-own approaches to managed reuse in an SPL as the general approach is well familiar to them and only the way operations are stored changes.

Mining of Cross-Tree Constraints Our Matador SPL approach is able to automatically identify recommendations for possible cross-tree constraints and relations between identified features. While there might be ways to identify further cross-tree constraints, we believe that any identified relations can only be considered as recommendations due to the extractive creation of SPLs using our Matador SPL approach. Main reason is that deriving cross-tree constraints from a subset of all possible variants (i.e., the existing input variants) cannot consider enough information to identify universally valid constraints. Thus, this approach will always derive cross-tree constraints that are either too restrictive or too permissive. As a result, we limited our recommendations to a subset of relations where we believe it is possible to derive sensible constraints that are actually helpful to developers. For example, identified mutually exclusive relations are indicators for features that cannot coexist due to potentially conflicting functionality. In addition, the requires relations between child features which are contained in the same variants as their parents are helpful to indicate functionality that might as well be merged into corresponding parent features as they always coexist.

8.3. Possible Future Application Areas

In this thesis, we presented semi-automatic migration of existing model variants to a delta-oriented SPL realization. The following section discusses potential future application areas beyond the scope of this thesis to provide initial ideas and elaborates on linked challenges as well as potentials.

Improvement of 150% Model Visualization A possible future direction is to investigate better forms of representation for information in 150% models. To the best of our knowledge, currently no efforts exist to visualize such low-level variability in a user-understandable way. Additional research in this area can help to improve the experience of users when working with 150% models. A general challenge with 150% models is the high variability introduced by a large number of variants. Thus, sensible approaches are needed to reduce this complexity by using a notation close to the original modeling notation (e.g., Matlab/Simulink). Similar to the visualization of models in their original

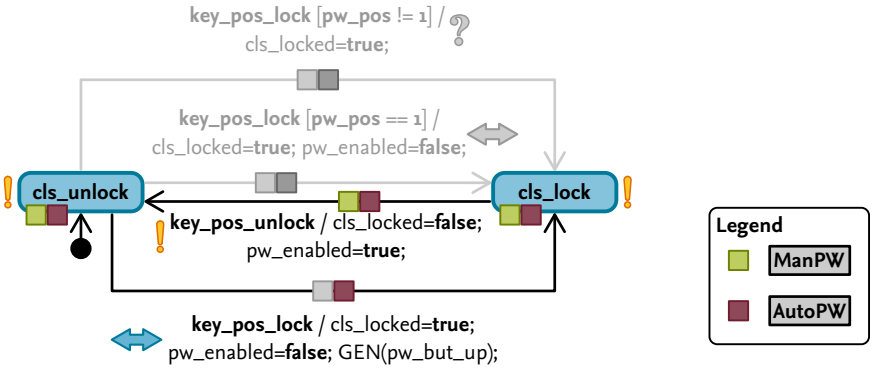


Figure 8.2.: A possible concept for a visualization of the 150% model in Figure 4.10 for the Central Locking System (CLS) variants from our running example in Figure 2.5.

notation a new 150% model visualization should exploit the hierarchical decomposition to hide model elements that are not relevant for the current task. Furthermore, concepts are needed to display for each of the model elements in which variants they are contained without overwhelming users with too much information. Using additional tooling to apply filtering on the 150% model, developers can be supported in efficiently analyzing model elements that are relevant for specific variants or have a direct relation (e.g., alternative groups).

In Figure 8.2, we show an initial concept for a corresponding visualization of the 150% model in Figure 4.10. Instead of using textual labels to annotate which variants comprise the model elements, we translate them to a color encoding. This way, relations between model variants are shown and can more easily be grasped by users as the differing colors visually support differentiation. Furthermore, we believe that reusing already known notations, such as the question marks, exclamation marks and double headed arrows, additionally support developers in understanding the relations. One feature indicated by the concept in Figure 8.2 is the filtering of the 150% model as, in this case, the user selected to only display all model elements that are present in variant AutoPW. All remaining model elements are grayed out to allow focusing on the selected variant. Further filtering could include reduction to model elements of a selected alternative group or search for model elements comprising a certain substring (e.g., a certain event in transitions).

In summary, we believe that our Family Mining approach provides a solid basis for detailed analysis of fine-grained variability information and that continued work on improving currently used visualizations can increase the usability of the already appreciated information.

Integration of Tooling in Development Environments Another insight gained from the expert interviews is that engineers wish for an integration of our approach with their standard development environment. Main reason is that switching between different tools during everyday work is a disturbance for the engineers' workflow. Thus, a tight integration with existing tooling is essential to support the acceptance of the provided solutions. Furthermore, an integrated solution also reduces the potentially required training period to apply the additional tooling. Overall, we believe that our developed solutions provide the necessary basis for such integration within existing development tools as the underlying algorithms are already well evaluated. Thus, companies can completely focus on identifying how to actually integrate the algorithms in their tools.

Application to Intertwined Development Languages The solution presented in this thesis focuses on generic variability mining solutions that can be applied to a variety of different languages. During our work on the Family Mining algorithms, we showed applicability of our approach to FBDs (i.e., a notation close to Matlab/Simulink) as part of the IEC 61131-3 standard [Int13] for automation control systems [HWL+14]. However, this standard is not limited to FBDs only, but also comprises other model-based languages (e.g., *sequential function charts*, a notation close to Petri nets) as well as textual languages (e.g., *structured text*, a notation close to Pascal). In addition, these different notations can be intertwined as it is, for example, possible to define the functionality of a block in FBDs using a description in structured text. Thus, to allow for complete variability mining of IEC 61131-3 software variants, additional research is necessary to identify all possible variability relations.

Nevertheless, we believe that we already provide the groundwork for such a holistic variability mining approach as our generic Family Mining with its tooling should be applicable with low effort to adapt our algorithms for the other block-based languages used by the IEC 61131-3 standard [Int13]. In addition, we showed in further research that it is possible to apply model-based solutions for variability mining of object-oriented source code, such as Java or C++ [WTS+16]. The presented algorithms use language-specific parsers to transform the analyzed source code into instances of a generic meta-model for different programming languages and apply variability mining to these generic representations. Thus, we believe that our solutions from [WTS+16] should be adaptable for other textual languages with low effort. The remaining challenge after an adaptation of the existing solutions is to combine the variability mining for block-based model variants with the variability mining for source code variants (e.g., to be able to compare *Function Blocks* in Matlab/Simulink, which allow definition of their functionality using C or C++ code). Most importantly, it is necessary to identify how to decide whether the functionality of blocks can be compared based on their atomic functions or whether it has to be compared using a source code comparison. Furthermore, question remains what happens in cases where one block using an atomic library function is compared with a block whose functionality was defined using a textual language.

Overall, we believe that our research in the area of variability mining of block-based languages and source code provides a solid basis for this continued work and helps to at least provide insights on possible approaches. In fact, our research on the Family Mining approach led to the *Reverse Engineering Design of Software Product Lines for Automation Technology (RED SPLAT)*[1] research project funded by the *Deutsche Forschungsgemeinschaft (DFG)*[2] – German Research Foundation. Focus is variability mining of models developed with the languages in the IEC 61131-3 standard [Int13].

Combination with Mining Approaches for other Abstraction Layers Another interesting research direction could be the combination of our Family Mining with variability mining for notations on other abstraction levels of developed systems. While we successfully demonstrated applicability of model-based variability mining for object-oriented source code [WTS+16] (i.e., a abstraction level below model-based languages), it can also be helpful to analyze systems on an even higher level of abstraction. As a result, we applied the idea of variability mining to descriptions of technical architectures in IT systems [WWS+17c]. These architectures comprise *layers* of systems (e.g., the hardware, operating system and application servers for executed software) and *tiers* allowing to define complex client-server applications. Using a variability mining specifically designed for the peculiarities of

[1] https://www.ais.mw.tum.de/en/research/current-research-projects/red-splat/
[2] http://www.dfg.de/

such technical architectures, we were able to identify variability across different technical architectures employed in companies (e.g., multiple servers running Java applications) [WWS+17c].

We believe that combining all three approaches, i.e., the low-level approach for variability mining of source code [WTS+16], the approach described in this thesis and the high-level approach for variability mining of technical architecture descriptions [WWS+17c], can be beneficial for companies. For instance, we were able to show that it is possible to automatically identify restructuring potential for IT systems to reduce corresponding maintenance and license costs [WWP+16, WWS+17a, WWS+17b, WWS+18]. As a result, it is possible to analyze complete systems including not only the developed software, but also their execution environment. On the one hand, it is possible to reduce the linked development and maintenance costs for software families by migrating a set of existing model and source code variants to corresponding SPLs using our described approaches. On the other hand, companies can also reduce the costs linked with variability present in architectures of their IT systems. A corresponding integrated approach of the different techniques might require additional research to ensure that all requirements for the developed software are still met by the underlying technical architectures when restructuring them. However, we believe that our variability mining approaches for the three levels of abstraction are a sensible basis for such a system.

Part IV.

Appendix

A Base Meta-Model

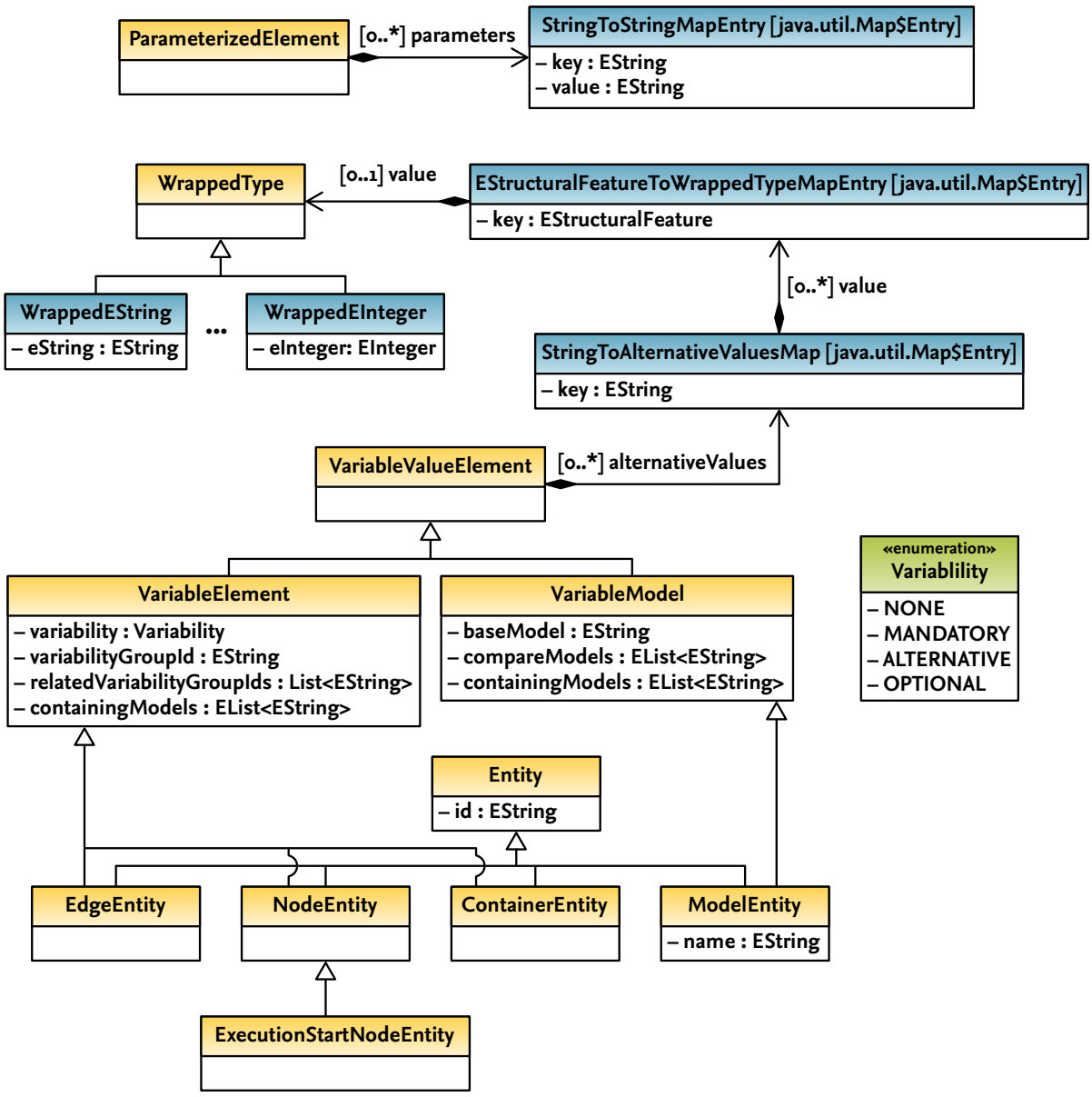


Figure A.1.: The base meta-model allowing to define variability-aware meta-models for FAMILY MINING.

```
Settings {
  modelName base
  modelUri "http://www.tu-bs.de/isf/familymining/base"
  modelPrefix base
  namespacePrefix de.tu_bs.cs.isf.familymining
  fileExtension base
  generateModelOnly
  representsBaseModel
}

Map StringToStringMapEntry {
  mapKey EString
  mapValue EString
}

Map EStructuralFeatureToWrappedTypeMapEntry {
  mapKey EStructuralFeature
  mapValue containment WrappedType
}

Map StringToAlternativeValuesMap {
  mapKey EString
  mapValue containment mapReference
    EStructuralFeatureToWrappedTypeMapEntry
}

abstract ClassEntity ParameterizedElement {
  mapReference parameters StringToStringMapEntry
}

abstract ClassEntity VariableValueElement {
  mapReference alternativeValues StringToAlternativeValuesMap
}

abstract ClassEntity VariableModel extends VariableValueElement {
  attribute baseModel, EString
  attribute compareModels, EString, [0..-1]
  attribute containingModels, EString, [0..-1]
}

abstract ClassEntity VariableElement extends VariableValueElement {
  attribute variabilityGroupId, EString
  attribute containingModels, EString, [0..-1]
  attribute relatedVariabilityGroupIds, EString, [0..-1]
  attribute variability, Variability
}

abstract ClassEntity Entity {
  identifier attribute id, EString, [1..1]
}


```

```
53 abstract ClassEntity ModelEntity extends Entity, VariableModel {
54   attribute name, EString, [1..1]
55 }
56
57 abstract ClassEntity ContainerEntity extends Entity, VariableElement
58
59 abstract ClassEntity NodeEntity extends Entity, VariableElement
60
61 abstract ClassEntity ExecutionStartNodeEntity extends NodeEntity
62
63 abstract ClassEntity EdgeEntity extends Entity, VariableElement
64
65 abstract ClassEntity WrappedType
66
67 ClassEntity WrappedEString extends WrappedType {
68   attribute wrappedEString, EString
69 }
70
71 // ...
72
73 ClassEntity WrappedEInteger extends WrappedType {
74   reference wrappedEInteger, EInt
75 }
76
77 Enum Variability {
78   NONE,
79   MANDATORY,
80   ALTERNATIVE,
81   OPTIONAL
82 }
```

Listing A.1: Vampire description for our base meta-model in Figure A.1.

B Vampire DSL

The following URI is used as a source in `EAnnotations` to annotate additional meta-model information that can be used by our Vampire DSL to generate artifacts from existing meta-models:

`http://www.tu-braunschweig.de/isf/familymining/MetaModelAnnotation`

In Table B.1, we show a mapping between the Vampire DSL and corresponding `EAnnotations`. Using these annotations in an existing meta-model allows to generate corresponding Family Mining artifacts from a Vampire description similar to Listing B.1.

```
1 Settings {
2   existingMetaModel "<path/to/existing/model.ecore>"
3   existingGenModel "<path/to/existing/model.genmodel>"
4   namespacePrefix de.tu_bs.cs.isf.familymining
5 }
```

Listing B.1: Vampire description to generate Family Mining extensions from an existing meta-model with `EAnnotations` following the mapping in Table B.1.

VAMPIRE DSL keyword	EAnnotation
Thresholds	
`mandatoryThreshold` with *`Double-Value`* in `Thresholds`	`MandatoryThreshold` → *`Double-Value`*
`optionalThreshold` with *`Double-Value`* in `Thresholds`	`OptionalThreshold` → *`Double-Value`*
Decision Wizard	
`nameBasedDecisionWizard` in `Settings`	`DecisionWizardGeneration` → `NameBased`
Entity Types	
`ModelEntity`	`EntityType` → `ModelEntity`
`ContainerEntity`	`EntityType` → `ContainerEntity`
`NodeEntity`	`EntityType` → `NodeEntity`
`ExecutionStartNodeEntity`	`EntityType` → `ExecutionStartNodeEntity`
`EdgeEntity`	`EntityType` → `EdgeEntity`
`Entity`	`EntityType` → `Entity`
Weights	
`subEntities` with *`Double-Value`* in `Weights`	`SubEntitiesWeight` → *`Entity Double-Value`* (, *`Entity Double-Value`*)
`incoming` with *`Double-Value`* in `Weights`	`IncomingWeight` → *`Double-Value`*
`outgoing` with *`Double-Value`* in `Weights`	`OutgoingWeight` → *`Double-Value`*
`Attribute` with *`Double-Value`* in `Weights`	`AttributeWeight` → *`Double-Value`*
`redirect` and *`Attribute`* with *`Double-Value`* in `Weights`	`RedirectWeight` → *`Double-Value`*
`sourceEntity` and *`Attributes`* with *`Double-Values`* in `Weights`	`SourceWeight` → *`Attribute Double-Value`* (, *`Attribute Double-Value`*)
`targetEntity` and *`Attributes`* with *`Double-Values`* in `Weights`	`TargetWeight` → *`Attribute Double-Value`* (, *`Attribute Double-Value`*)
`typeCheck` and *`Entity`* with *`Double-Value`* in `Weights`	`TypeCheckWeight` → *`Double-Value`*
String Comparison Algorithm	
`equals`	`StringComparisonApproach` → `Equals`

Table B.1.: Mapping between VAMPIRE keywords and corresponding `EAnnotations`.

C Variability Mining of Statecharts

In this chapter, we show details of our current Family Mining implementation for the statechart notation by Harel [Har87], the UML standard by the OMG [Obj15] and the tool-specific notations of The MathWorks Stateflow, ETAS ASCET, IBM Rational Rhapsody, Esterel Technologies SCADE Suite and Yakindu.
The shown meta-model and metric were used in [WSS16] and [Wil14].

C.1. Meta-Model for Statecharts

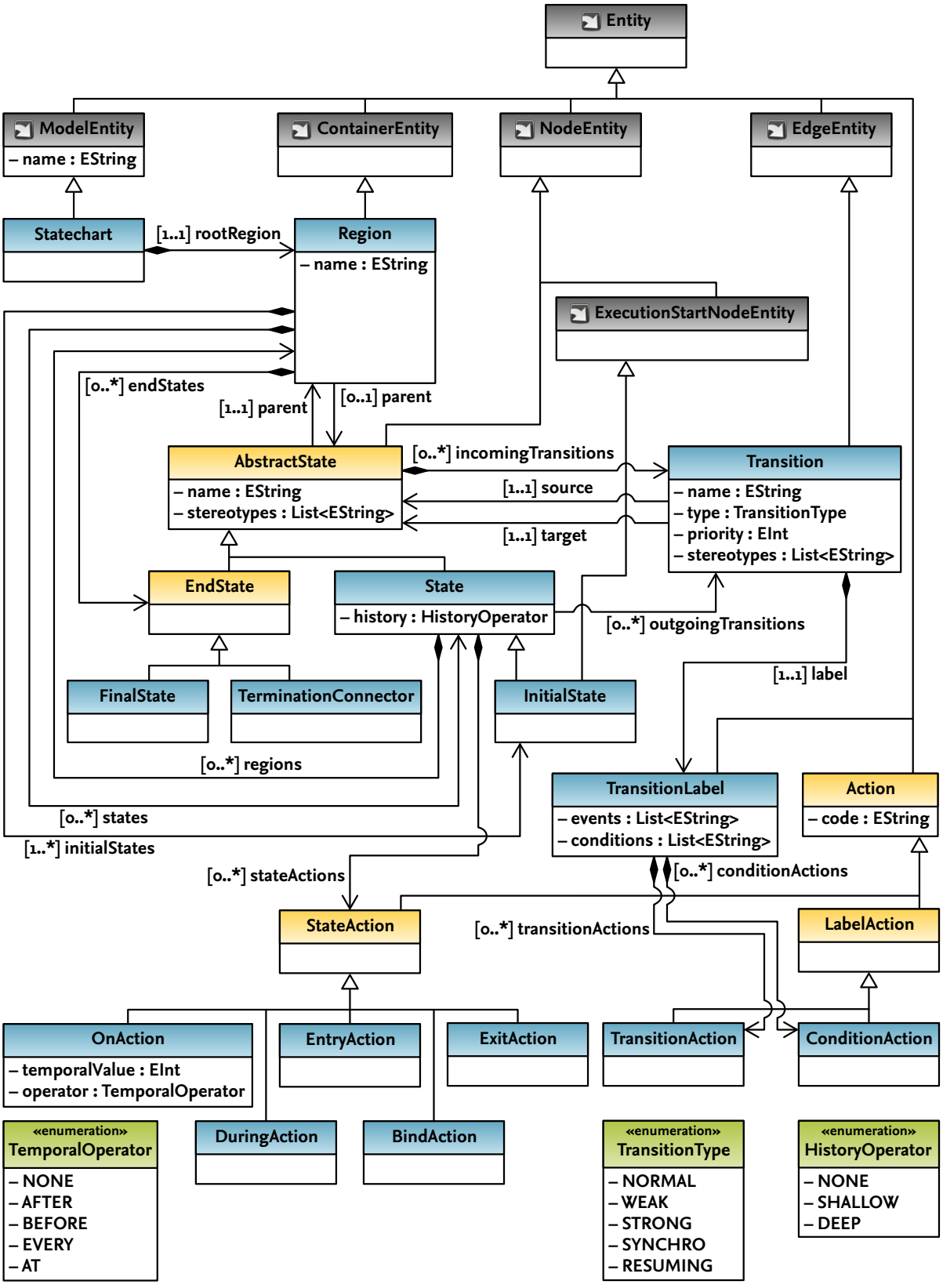


Figure C.1.: Meta-model used for Family Mining of statecharts.

On the following two pages, we show the Vampire DSL description to generate the statechart metamodel in Figure C.1.

```
Settings {
  modelName stateoriented
  modelUri "http://www.tu-bs.de/isf/familymining/stateoriented"
  modelPrefix stateoriented
  namespacePrefix de.tu_bs.cs.isf.familymining
  fileExtension statechart
}

ModelEntity Statechart subEntities Region rootRegion [1..1]

abstract NodeEntity AbstractState
  parentEntity Region parent incoming Transition
{
  attribute name, EString, [1..1]
  attribute stereotypes, EString, [0..-1]
}

NodeEntity State subEntities Region regions [0..-1]
  extends AbstractState outgoing Transition
{
  attribute history, HistoryOperator, [1..1]
  containment reference actions, StateAction, [0..-1]
}

abstract NodeEntity EndState extends AbstractState

NodeEntity FinalState extends EndState

NodeEntity TerminationConnector extends EndState

ExecutionStartNodeEntity InitialState extends State

ContainerEntity Region parentEntity AbstractState parent [0..1]
  subEntities InitialState initialStates [1..-1],
    State states, EndState endStates
{
  attribute name, EString, [1..1]
}

EdgeEntity Transition sourceEntity AbstractState source
  targetEntity AbstractState target
{
  attribute name, EString
  attribute priority, EInt
  attribute type, TransitionType, [1..1]
  attribute stereotypes, EString, [0..-1]
  containment reference label, TransitionLabel, [1..1]
}

abstract Entity Action {
  attribute code, EString, [1..1]
}
```

```
53  Entity TransitionLabel {
54    attribute events, EString, [0..-1]
55    attribute conditions, EString, [0..-1]
56    containment reference conditionActions, ConditionAction, [0..-1]
57    containment reference transitionActions,
58        TransitionAction, [0..-1]
59  }
60
61  abstract Entity StateAction extends Action
62
63  Entity EntryAction extends StateAction
64
65  Entity ExitAction extends StateAction
66
67  Entity DuringAction extends StateAction
68
69  Entity BindAction extends StateAction
70
71  Entity OnAction extends StateAction {
72    attribute temporalValue, EInt, [1..1]
73    attribute temporalOperator, TemporalOperator, [1..1]
74  }
75
76  abstract Entity LabelAction extends Action
77
78  Entity ConditionAction extends LabelAction
79
80  Entity TransitionAction extends LabelAction
81
82  Enum HistoryOperator {
83    NONE,
84    SHALLOW,
85    DEEP
86  }
87
88  Enum TemporalOperator {
89    NONE,
90    AFTER,
91    BEFORE,
92    EVERY,
93    AT
94  }
95
96  Enum TransitionType {
97    NORMAL,
98    WEAK,
99    STRONG,
100   SYNCHRO,
101   RESUMING
102 }
```

Listing C.1: Vampire description for the statecharts meta-model in Figure C.1.

C.2. Metric for Statecharts

The following table shows a concrete metric used for Family Mining of statechart variants [Wil14]. Properties with a weight of 0% were not present in the analyzed models and, thus, deactivated for the executed comparisons.

Property	Weight
States – Static Properties	
name	20%
initial state	20%
end state	0%
parallel state	10%
hierarchy distance	25%
neighborhood	25%
stereotype	0%
States – Dynamic Properties	
history state	0%
dependent on event	40%
triggered actions	40%
events triggering change	20%
States – Neighborhood Weighting	
neighbor similarity	50%
neighbor interface similarity	50%
Neighbor State Properties	
name	20%
actions	80%

Property	Weight
Transitions – Static Properties	
names	100%
stereotype	0%
Transitions – Dynamic Properties	
transition label	100%
type	0%
priority	0%
Transition Label Properties	
events	40%
conditions	30%
condition actions	0%
transition actions	30%
Region Weighting	
sub states	50%
sub transitions	50%
State Weighting	
static properties	50%
dynamic properties	50%
Transition Weighting	
static properties	20%
dynamic properties	80%

Table C.1.: Concrete metric used for Family Mining of statecharts.

D Variability Mining of Matlab/Simulink Models and Function Block Diagrams

In this chapter, we show details of our current Family Mining implementation for Matlab/Simulink models that was also used for the analysis of FBDs as part of the IEC 61131-3 standard [Int13]. The shown meta-model and metric were used in [SWS+17] and [HWL+14].

D.1. Meta-Model for Matlab/Simulink Models and Function Block Diagrams

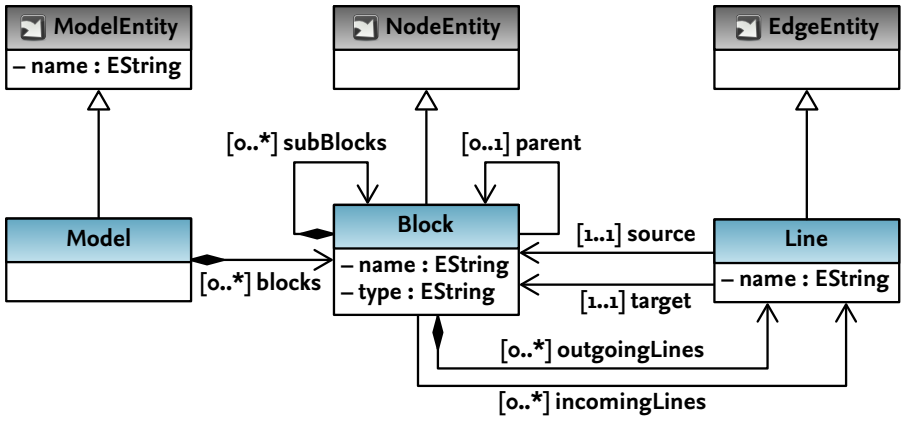


Figure D.1.: The meta-model used for Family Mining of Matlab/Simulink variants and IEC 61131-3 FBD [Int13] variants).

```
1  Settings {
2    modelName blockoriented
3    modelUri "http://www.tu-bs.de/isf/familymining/blockoriented"
4    modelPrefix blockoriented
5    namespacePrefix de.tu_bs.cs.isf.familymining
6    fileExtension blockoriented
7    nameBasedDecisionWizard
8  }
9
10 parameterizable ModelEntity Model subEntities Block blocks
11
12 parameterizable NodeEntity Block parentEntity Block parent
13   subEntities Block incoming Line outgoing Line
14 {
15   attribute name, EString, [1..1]
16   attribute type, EString, [1..1]
17 }
18
19 parameterizable EdgeEntity Line
20   sourceEntity Block source targetEntity Block target
21 {
22   attribute name, EString, [0..1]
23 }
```

Listing D.1: Vampire description for the Matlab/Simulink and IEC 61131-3 FBD [Int13] meta-model in Figure D.1.

D.2. Metric for Matlab/Simulink Models and Function Block Diagrams

Property	Weight
name	5%
function	75%
Inports	
• number of inports	5%
• number of matching inport functions	5%
Outports	
• number of outports	5%
• number of matching outport functions	5%

Table D.1.: Concrete metric used for Family Mining of Matlab/Simulink variants and IEC 61131-3 FBD [Int13] variants).

E Refactoring Operators Applied to the Generated SPL

Operator: Merge Into (Shallow)		
ManPW	merged into	BCS_ManPW
AutoPW	merged into	BCS_AutoPW
FP	merged into	BCS_FP
EM_Heat	merged into	BCS_EM_Heat
AS	merged into	BCS_AS
EM	merged into	BCS_EM
CLS	merged into	BCS_CLS
RCK	merged into	BCS_RCK
LED	merged into	BCS_LED
LED_AS_Alarm	merged into	LED_AS
LED_AS_Alarm_Detected	merged into	LED_AS
LED_AS_Active	merged into	LED_AS
HMI	merged into	BCS_HMI
Operator: Rename Feature		
BCS_ManPW	renamed to	ManPW
BCS_AutoPW	renamed to	AutoPW
BCS_FP	renamed to	FP
BCS_EM_Heat	renamed to	Heatable
BCS_AS	renamed to	AS
BCS_EM	renamed to	EM
BCS_CLS	renamed to	CLS
BCS_RCK	renamed to	RCK
BCS_LED	renamed to	LED
LED_AS	renamed to	LED AS
LED_EM_Heat	renamed to	LED Heatable

Table E.1.: Part one of the refactoring operators applied to the feature model generated by the MATADOR SPL approach in Figure 5.2 to derive the final SPL feature model in Figure 5.4.

Operator: Rename Feature		
BCS_HMI	renamed to	HMI
CLS_V1_V4_V6	renamed to	CLS ManPW
CLS_V3_V7	renamed to	CLS AutoPW
Operator: Add Feature		
Door System	added below	BCS
Security	added below	BCS
PW	added below	Door System
Operator: Change Variability		
Door System	changed to	*mandatory*
PW	changed to	*mandatory*
Operator: Move Feature		
ManPW	moved below	PW
AutoPW	moved below	PW
FP	moved below	PW
EM	moved below	Door System
AS	moved below	Security
RCK	moved below	Security
CLS	moved below	Security
Heatable	moved below	EM
CLS_V2	moved below	RCK
FP_V1_V4_V5_V6	moved below	ManPW
FP_V2_V3_V7	moved below	AutoPW
Operator: Merge Into (Shallow)		
CLS_V2	merged into	RCK
FP_V1_V4_V5_V6	merged into	ManPW
FP_V2_V3_V7	merged into	AutoPW
Operator: Remove with Artifacts		
HMI_Controller	removed	
Operator: Create Alternative		
ManPW and AutoPW	set alternative	
Operator: Create Or		
LED Heatable and LED AS	set or	

Table E.2.: Part two of the refactoring operators applied to the feature model generated by the MATADOR SPL approach in Figure 5.2 to derive the final SPL feature model in Figure 5.4.

F Extension Points of the Family Mining Framework

The following list gives a detailed overview of the Family Mining framework extension points shown in Figure 6.3 and corresponding data structures provided by the framework to support their implementation as well as realized default implementations.

Importer Extension Point

Description: This extension point allows definition of algorithms to realize language-specific import of models from their original notation into a corresponding meta-model.

Input: A model that should be imported.

Output: A model container storing the imported model.

Supporting Structures: `ModelContainer` allowing handling of imported models during processing by the Family Mining framework independent of their type.

Default Implementation: Generic importer for EMF Ecore-based models stored in XMI files.

Automatic File Sorting Extension Point

Description: This extension point allows definition of algorithms to automatically sort models prior processing by subsequent steps.

Input: A list of model files.

Output: A list of model files sorted according to the executed algorithm.

Default Implementation: An algorithm sorting the files in alphabetical order based on their names (either ascending or descending).

Comparison Extension Point

Description: This extension point allows definition of algorithms to compare models using *pairwise* or *n-way* algorithms.

Input: A list of model containers storing imported models.

Output: A list of comparison elements storing the comparison results.

Supporting Structures:

- `ComparisonElement` allowing to store an arbitrary number of compared model elements together with a similarity value.
- Classes to store weighted similarity values with all intermediate steps to allow transparency for calculation results.

Default Implementation: The generic and adaptable Family Mining comparison algorithm discussed in Chapter 4.

Metric Extension Point

Description: This extension point allows implementation of concrete comparisons between model elements using *pairwise* or *n-way* algorithms. Furthermore, methods can be implemented for the categorization of variability relations into mandatory, alternative and optional parts during merging.

Input: A list of model elements that should be compared.

Output: A comparison element storing the compared model elements together with a calculated similarity value.

Supporting Structures:

- `ComparisonElement` allowing to store an arbitrary number of compared model elements together with a similarity value.
- Classes to store weighted similarity values with all intermediate steps to allow transparency for calculation results.

String Comparison Algorithm Extension Point

Description: This extension point allows implementation of algorithms to compare strings.

Input: Two strings that should be compared.

Output: A double value indicating the strings similarity according to the executed algorithm.

Default Implementation: An implementation of the Levenshtein Distance algorithm [Lev66].

Match Extension Point

Description: This extension point allows definition of algorithms to match comparison elements to remove ambiguities that might exist due to multiple possible relations between specific model elements.

Input: A list of comparison elements with potential ambiguities.

Output: A list of matched comparison elements that contain distinct relations between compared models.

Default Implementation: The generic Family Mining match algorithm discussed in Chapter 4.

Decision Wizard Extension Point

Description: This extension point allows implementation of manual or automatic algorithms to resolve conflicts during matching of comparison elements.

Input: A list of ambiguous comparison elements.

Output: A matched comparison element that was selected as a resolution.

Supporting Structures: Classes to support implementation of GUI elements to allow manual selection of resolutions.

Merge Extension Point

Description: This extension point allows implementation of iterative *pairwise* merging or direct *n-way* merging of identified variability relations in a single model.

Input: A list of matched comparison elements that contain distinct relations between compared models together with the model containers storing their imported models.

Output: A merged 150% model storing the identified variability relations.

Exporter Extension Point

Description: This extension point allows user-specified export of merged 150% models (e.g., reports or a corresponding representation in the original input notation).

Input: A merged 150% model storing the identified variability relations.

Output: An export of the 150% model to a user-specified notation.

Default Implementation:

- A generic EMF exporter storing the 150% model in a XMI file corresponding to the used meta-model.
- The Matador SPL approach to generate a corresponding delta-oriented SPL (cf. Chapter 5).

G Questionnaire for the Expert Interviews

1. **General Expectations**
   a) *What are your overall expectations regarding our* Family Mining *approach?*
   This questions aims to get an idea of the very basic expectations model engineers have regarding our Family Mining approach.
   b) *What are your expectations regarding the output of our approach?*
   The output of the Family Mining approach is the final 150% model. Within this model, variability information is annotated. This questions aims to get an idea of the demands a model engineer has regarding the output of our approach so that the information can effectively be used for the engineer's own work.
2. **Fields of Application**
   a) *Using the* Family Mining *approach allows you to analyze the variability of two or more than two models. Regarding your current work, what are the specific use cases where either one of the two scenarios might be useful to you?*
   By offering the capability of analyzing more than two models, the Family Mining approach can be associated with techniques evolving around the generation of SPLs. With this question, we want to find out whether this scenario is of interest to the model engineer or if the primary focus is on the variability analysis of only two models.
   b) *Regarding your current work, what specific scenarios exist where the* Family Mining *approach can be of an assistance to you?*
   This question aims at naming specific use cases in which the engineer would see our approach as applicable and useful.
   c) *What is more important to you, shorter runtime or a higher precision of the produced results? Please explain your answer.*
   With this question, we want to find out about the preferences of a model engineer in an industrial environment to further improve our approach based on the given feedback.
   d) *Is there a situation in which you might be willing to reduce the overall runtime and, thus, potentially the precision of the produced results also?*
   With this questions, we want to identify potential use cases which where not previously mentioned by the model engineer.

3. **Usability of Results**

   a) *Based on your expectations regarding the output, where do you see room for improving the results to understand and effectively use them?*
   With this question, we want the engineer to identify room for improvement regarding both the contained information of the output and the form of representation itself.

   b) *Regarding the produced output, what specific use cases exist for using the output for your current work?*
   With this question, we aim to find out, to what extent the produced output can be used by the model engineer.

   c) *Rating the produced output from 1 (bad) to 5 (very good), to what extent does the output reflect your perception of a well-documented variability?*
   With this question, we want to get a quantified impression on overall quality of the produced output.

Bibliography

[AAC13] E. P. Antony, M. H. Alalfi, and J. R. Cordy. "An Approach to Clone Detection in Behavioural Models". In: *Proc. of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 472–476.

[Aal16] W. M. P. van der Aalst. *Process Mining: Data Science in Action*. Springer, 2016.

[ABK+13] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[ACC+11] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. "Reverse Engineering Architectural Feature Models". In: *Proc. of the European Conference on Software Architecture (ECSA)*. Vol. 6903. Lecture Notes in Computer Science. Springer, 2011, pp. 220–235.

[ACC+14] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. "Extraction and Evolution of Architectural Variability Models in Plugin-based Systems". In: *Software & Systems Modeling* 13.4 (Oct. 2014), pp. 1367–1394.

[ACD+12a] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson. "Models are Code too: Near-Miss Clone Detection for Simulink Models". In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 295–304.

[ACD+12b] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson. "Near-Miss Model Clone Detection for Simulink Models". In: *Proc. of the Intl. Workshop on Software Clones (IWSC)*. IEEE, 2012, pp. 78–79.

[ACD14] M. H. Alalfi, J. R. Cordy, and T. R. Dean. "Analysis and Clustering of Model Clones: An Automotive Industrial Experience". In: *Proc. of the Intl. Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 375–378.

[ACN+08] V. Alves, F. Calheiros, V. Nepomuceno, A. Menezes, S. Soares, and P. Borba. "FLiP: Managing Software Product Line Extraction and Reaction with Aspects". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. IEEE, 2008, pp. 354–354.

[ACP+12] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire. "On Extracting Feature Models from Product Descriptions". In: *Proc. of the Intl. Workshop on Variability Modeling in Software-Intensive Systems (VaMoS)*. ACM, 2012, pp. 45–54.

[ACS+12] N. Andersen, K. Czarnecki, S. She, and A. Wasowski. "Efficient Synthesis of Feature Models". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 2012, pp. 106–115.

[AFH+15] L. Arcega, J. Font, Ø. Haugen, and C. Cetina. "Leveraging Models at Run-Time to Retrieve Information for Feature Location". In: *Proc. of the Intl. Workshop on MoDELS@-run.time*. CEUR-WS.org, 2015, pp. 51–60.

[AFH+16a] L. Arcega, J. Font, Ø. Haugen, and C. Cetina. "An Infrastructure for Generating Runtime Model Traces for Maintenance Tasks". In: *Proc. of the Intl. Workshop on MoDELS@run.time.* CEUR-WS.org, 2016, pp. 35–42.

[AFH+16b] L. Arcega, J. Font, Ø. Haugen, and C. Cetina. "Feature Location Through the Combination of Run-Time Architecture Models and Information Retrieval". In: *Proc. of the Intl. Conference on System Analysis and Modeling (SAM).* Vol. 9959. Lecture Notes in Computer Science. Springer, 2016, pp. 180–195.

[AFH+17] L. Arcega, J. Font, Ø. Haugen, and C. Cetina. "On the Influence of Models at Run-Time Traces in Dynamic Feature Location". In: *Proc. of the European Conference on Modeling Foundations and Applications (ECMFA).* Vol. 10376. Lecture Notes in Computer Science. Springer, 2017, pp. 90–105.

[AGD14] N. Assy, W. Gaaloul, and B. Defude. "Mining Configurable Process Fragments for Business Process Design". In: *Proc. of the Intl. Conference on Design Science Research in Information Systems (DESRIST).* Vol. 8463. Lecture Notes in Computer Science. Springer, 2014, pp. 209–224.

[AGM+06] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. "Refactoring Product Lines". In: *Proc. of the Intl. Conference on Generative Programming and Component Engineering (GPCE).* ACM, 2006, pp. 201–210.

[AHS+14] R. Al-Msie'deen, M. Huchard, A.-D. Seriai, C. Urtado, and S. Vauttier. "Automatic Documentation of [Mined] Feature Implementations from Source Code Elements and Use-Case Diagrams with the REVPLINE Approach". In: *International Journal of Software Engineering and Knowledge Engineering* 24.10 (2014), pp. 1413–1438.

[AJB+14] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wasowski, and I. Schaefer. "Flexible Product Line Engineering with a Virtual Platform". In: *Proc. of the Intl. Conference on Software Engineering (ICSE).* ACM, 2014, pp. 532–535.

[ALL+15] W. K. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed. "Extracting Variability-Safe Feature Models from Source Code Dependencies in System Variants". In: *Proc. of the Intl. Genetic and Evolutionary Computation Conference (GECCO).* ACM, 2015, pp. 1303–1310.

[ALL+17a] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed. "Multi-Objective Reverse Engineering of Variability-Safe Feature Models Based on Code Dependencies of System Variants". In: *Empirical Software Engineering* 22.4 (Aug. 2017), pp. 1763–1794.

[ALL+17b] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed. "Reengineering Legacy Applications into Software Product Lines: A Systematic Mapping". In: *Empirical Software Engineering* 22.6 (Dec. 2017), pp. 2972–3016.

[AMC+05] V. Alves, P. Matos, L. Cole, P. Borba, and G. Ramalho. "Extracting and Evolving Mobile Games Product Lines". In: *Proc. of the Intl. Software Product Line Conference (SPLC).* Vol. 3714. Lecture Notes in Computer Science. Springer, 2005, pp. 70–81.

[AMC+07] V. Alves, P. Matos, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho. "Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming". In: *Proc. of the Intl. Conference on Aspect-Oriented Software Development (AOSD)*. Vol. 4640. Lecture Notes in Computer Science. Springer, 2007, pp. 117–142.

[ANC+12] V. Anwikar, R. Naik, A. Contractor, and H. Makkapati. "Domain-driven Technique for Functionality Identification in Source Code". In: *ACM SIGSOFT Software Engineering Notes* 37.3 (May 2012), pp. 1–8.

[AP03] M. Alanen and I. Porres. "Difference and Union of Models". In: *Proc. of the Intl. Conference on the Unified Modeling Language (UML)*. Vol. 2863. Lecture Notes in Computer Science. Springer, 2003, pp. 2–17.

[ARS+14] M. H. Alalfi, E. J. Rapos, A. Stevenson, M. Stephan, T. R. Dean, and J. R. Cordy. "Semi-Automatic Identification and Representation of Subsystem Variability in Simulink Models". In: *Proc. of the Intl. Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 486–490.

[AS96] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. Second Edition. The MIT Press, 1996.

[ASH+13] *Mining Features from the Object-Oriented Source Code of a Collection of Software Variants Using Formal Concept Analysis and Latent Semantic Indexing*. World Scientific Publishing, 2013, pp. 244–249.

[ASH11] B. Al-Batran, B. Schätz, and B. Hummel. "Semantic Clone Detection for Model-Based Development of Embedded Systems". In: *Proc. of the Intl. Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 6981. Lecture Notes in Computer Science. Springer, 2011, pp. 258–272.

[ASW09] K. Altmanninger, M. Seidl, and M. Wimmer. "A Survey on Model Versioning Approaches". In: *International Journal of Web Information Systems* 5.3 (2009), pp. 271–304.

[AV14] W. K. G. Assunção and S. R. Vergilio. "Feature Location for Software Product Line Migration: A Mapping Study". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 2014, pp. 52–59.

[BAB+16] G. Bécan, M. Acher, B. Baudry, and S. B. Nasr. "Breathing Ontological Knowledge into Feature Model Synthesis: An Empirical Study". In: *Empirical Software Engineering* 21.4 (Aug. 2016), pp. 1794–1841.

[Bab16] Ö. Babur. "Statistical Analysis of Large Sets of Models". In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. ACM, 2016, pp. 888–891.

[Bab18] Ö. Babur. "Clone Detection for Ecore Metamodels Using N-grams". In: *Proc. of the Intl. Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SCITEPRESS, 2018, pp. 411–419.

[Bak92] B. S. Baker. "A Program for Identifying Duplicated Code". In: Springer, 1992, pp. 49–57.

[Bak95] B. S. Baker. "On Finding Duplication and Near-duplication in Large Software Systems". In: *Proc. of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 1995, pp. 86–95.

[Bato4] D. Batory. "Feature-Oriented Programming and the AHEAD Tool Suite". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 2004, pp. 702–703.

[Bato5] D. Batory. "Feature Models, Grammars, and Propositional Formulas". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. Vol. 3714. Lecture Notes in Computer Science. Springer, 2005, pp. 7–20.

[BBA+14] G. Bécan, S. Ben Nasr, M. Acher, and B. Baudry. "WebFML: Synthesizing Feature Models Everywhere". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 2014, pp. 112–116.

[BBB+14] B. Bislimovska, A. Bozzon, M. Brambilla, and P. Fraternali. "Textual and Content-Based Search in Repositories of Web Application Models". In: *ACM Transactions on the Web (TWEB)* 8.2 (Mar. 2014), 11:1–11:47.

[BBG+15] G. Bécan, R. Behjati, A. Gotlieb, and M. Acher. "Synthesis of Attributed Feature Models from Product Descriptions". In: *Proc. of the Intl. Conference on Software Product Line (SPLC)*. ACM, 2015, pp. 1–10.

[BC16] M. Boubakir and A. Chaoui. "A Pairwise Approach for Model Merging". In: *Proc. of the Intl. Symposium on Modelling and Implementation of Complex Systems (MISC)*. Vol. 1. Lecture Notes in Networks and Systems. Springer, 2016, pp. 327–340.

[BC17] Ö. Babur and L. Cleophas. "Using n-grams for the Automated Clustering of Structural Models". In: *Proc. of the Intl. Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*. Vol. 10139. Lecture Notes in Computer Science. Springer, 2017, pp. 510–524.

[BCB16] Ö. Babur, L. Cleophas, and M. van den Brand. "Hierarchical Clustering of Metamodels for Comparative Analysis and Visualization". In: *Proc. of the European Conference on Modeling Foundations and Applications (ECMFA)*. Vol. 9764. Lecture Notes in Computer Science. Springer, 2016, pp. 3–18.

[BCB18] Ö. Babur, L. Cleophas, and M. van den Brand. "Towards Distributed Model Analytics with Apache Spark". In: *Proc. of the Intl. Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SCITEPRESS, 2018, pp. 767–772.

[BCK03] R. Buhrdorf, D. Churchett, and C. W. Krueger. "Salion's Experience with a Reactive Software Product Line Approach". In: *Proc. of the Intl. Workshop on Software Product-Family Engineering (PFE)*. Vol. 3014. Lecture Notes in Computer Science. Springer, 2003, pp. 317–322.

[BCR+01] D. Binkley, R. Capellini, R. Raszewski, and C. Smith. "An Implementation of and Experiment with Semantic Differencing". In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 2001, pp. 82–91.

[BCV+16] Ö. Babur, L. Cleophas, T. Verhoeff, and M. van den Brand. "Towards Statistical Comparison and Analysis of Models". In: *Proc. of the Intl. Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SCITEPRESS, 2016, pp. 361–367.

[BCW11] K. Bak, K. Czarnecki, and A. Wasowski. "Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled". In: *Proc. of the Intl. Conference on Software Language Engineering (SLE)*. Vol. 6563. Lecture Notes in Computer Science. Springer, 2011, pp. 102–122.

[BDD+16] F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. "Automated Clustering of Metamodel Repositories". In: *Proc. of the Intl. Conference on Advanced Information Systems Engineering (CAiSE)*. Vol. 9694. Lecture Notes in Computer Science. Springer, 2016, pp. 342–358.

[BDF+08] I. Barone, A. De Lucia, F. Fasano, E. Rullo, G. Scanniello, and G. Tortora. "COMOVER: Concurrent Model Versioning". In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 2008, pp. 462–463.

[BDK92] P. Buneman, S. Davidson, and A. Kosky. "Theoretical Aspects of Schema Merging". In: *Proc. of the Intl. Conference on Extending Database Technology (EDBT)*. Vol. 580. Lecture Notes in Computer Science. Springer, 1992, pp. 152–167.

[Beu08] D. Beuche. "Modeling and Building Software Product Lines with Pure::Variants". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. IEEE, 2008, pp. 358–358.

[BKA+07] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. "Comparison and Evaluation of Clone Detection Tools". In: *IEEE Transactions on Software Engineering (TSE)* 33.9 (Sept. 2007), pp. 577–591.

[BKL+16] J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr. "Reasoning About Product-line Evolution Using Complex Feature Model Differences". In: *Automated Software Engineering* 23.4 (Dec. 2016), pp. 687–733.

[BKP+07] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann. "Engineering Automotive Software". In: *Proceedings of the IEEE* 95.2 (Feb. 2007), pp. 356–373.

[BLC16] M. Ballarín, R. Lapeña, and C. Cetina. "Leveraging Feature Location to Extract the Clone-and-Own Relationships of a Family of Software Products". In: *Proc. of the Intl. Conference on Software Reuse (ICSR)*. Vol. 9679. Lecture Notes in Computer Science. Springer, 2016, pp. 215–230.

[BOJ04] M. Beine, R. Otterbach, and M. Jungmann. "Development of Safety-Critical Software Using Automatic Code Generation". In: *Society of Automotive Engineers (SAE) World Congress & Exhibition*. Society of Automotive Engineers (SAE) International, Mar. 2004.

[Bos00] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press / Addison-Wesley, 2000.

[BP08] C. Brun and A. Pierantonio. "Model Differences in the Eclipse Modelling Framework". In: *UPGRADE: The European Journal for the Informatics Professional* IX.2 (2008), pp. 29–34.

[BRN+13] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski. "A Survey of Variability Modeling in Industrial Practice". In: *Proc. of the Intl. Workshop on Variability Modeling in Software-Intensive Systems (VaMoS)*. ACM, 2013, 7:1–7:8.

[Bro06] M. Broy. "Challenges in Automotive Software Engineering". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. ACM, 2006, pp. 33–42.

[BSE+16] O. Berreteaga, G. Sagardui, L. Etxeberria, U. Markiegi, and X. Perez. "Delta Rhapsody". In: *International Council on Systems Engineering (INCOSE) International Symposium* 26.1 (2016), pp. 25–41.

[BSL+10] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. "Variability Modeling in the Real: A Perspective from the Operating Systems Domain". In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. ACM, 2010, pp. 73–82.

[Buf95] J. Buffenbarger. "Syntactic Software Merging". In: *Software Configuration Management*. Vol. 1005. Lecture Notes in Computer Science. Springer, 1995, pp. 153–172.

[BYM+98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. "Clone Detection Using Abstract Syntax Trees". In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 1998, pp. 368–377.

[CA05] K. Czarnecki and M. Antkiewicz. "Mapping Features to Models: A Template Approach Based on Superimposed Variants". In: *Proc. of the Intl. Conference on Generative Programming and Component Engineering (GPCE)*. Vol. 3676. Lecture Notes in Computer Science. Springer, 2005, pp. 422–437.

[CCW+01] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. "CVSSearch: Searching Through Source Code using CVS Comments". In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 2001, pp. 364–373.

[CDA14] J. R. Cordy, T. R. Dean, and M. H. Alalfi. "Clone Detection in MATLAB Stateflow Models". In: *Proc. of the Intl. Workshop on Software Clones (IWSC)*. Vol. 63. Electronic Communications of the European Association of Software Science and Technology (EASST), 2014.

[CDA16] J. Chen, T. R. Dean, and M. H. Alalfi. "Clone Detection in MATLAB Stateflow Models". In: *Software Quality Journal* 24.4 (Dec. 2016), pp. 917–946.

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CH06] K. Czarnecki and S. Helsen. "Feature-based Survey of Model Transformation Approaches". In: *IBM Systems Journal* 45.3 (July 2006), pp. 621–645.

[CLR+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. Third Edition. The MIT Press, 2009.

[CN01] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[Cor06] J. R. Cordy. "The TXL Source Transformation Language". In: *Science of Computer Programming* 61.3 (Aug. 2006), pp. 190–210.

[Cor13] J. R. Cordy. "Submodel Pattern Extraction for Simulink Models". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 2013, pp. 7–10.

[CR11] J. R. Cordy and C. K. Roy. "The NiCad Clone Detector". In: *Proc. of the Intl. Conference on Program Comprehension (ICPC)*. IEEE, 2011, pp. 219–220.

[CRG+96] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. "Change Detection in Hierarchically Structured Information". In: *Proc. of the Intl. Conference on Management of Data (MOD)*. ACM, 1996, pp. 493–504.

[CSW08] K. Czarnecki, S. She, and A. Wasowski. "Sample Spaces and Feature Models: There and Back Again". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. IEEE, 2008, pp. 22–31.

[CW03] R. Conradi and A. I. Wang. *Empirical Methods and Studies in Software Engineering: Experiences from ESERNET*. Springer, 2003.

[CW07] K. Czarnecki and A. Wasowski. "Feature Diagrams and Logics: There and Back Again". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. IEEE, 2007, pp. 23–34.

[CW98] R. Conradi and B. Westfechtel. "Version Models for Software Configuration Management". In: *ACM Computing Surveys (CSUR)* 30.2 (June 1998), pp. 232–282.

[CZZ+05] K. Chen, W. Zhang, H. Zhao, and H. Mei. "An Approach to Constructing Feature Models Based on Requirements Clustering". In: *Proc. of the Intl. Conference on Requirements Engineering (RE)*. IEEE, 2005, pp. 31–40.

[DDD+11] R. Dijkman, M. Dumas, B. van Dongen, R. Käärik, and J. Mendling. "Similarity of Business Process Models: Metrics and Evaluation". In: *Information Systems* 36.2 (2011), pp. 498–516.

[DDF02] G. A. Di Lucca, M. Di Penta, and A. R. Fasolino. "An Approach to Identify Duplicated Web Pages". In: *Proc. of the Intl. Computer Software and Applications Conference (COMPSAC)*. IEEE, 2002, pp. 481–486.

[DDH+13] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans. "Feature Model Extraction from Large Collections of Informal Product Descriptions". In: *Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013, pp. 290–300.

[DDP16] N. Dintzner, A. van Deursen, and M. Pinzger. "FEVER: Extracting Feature-oriented Changes from Commits". In: *Prof. of the Intl. Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 85–96.

[DDP18] N. Dintzner, A. van Deursen, and M. Pinzger. "FEVER: An Approach to Analyze Feature-Oriented Changes and Artefact Co-Evolution in Highly Configurable Systems". In: *Empirical Software Engineering* 23.2 (Apr. 2018), pp. 905–952.

[DFS+09] A. De Lucia, F. Fasano, G. Scanniello, and G. Tortora. "Concurrent Fine-Grained Versioning of UML Models". In: *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2009, pp. 89–98.

[DHJ+08] F. Deißenböck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. "Clone Detection in Automotive Model-based Development". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 2008, pp. 603–612.

[DHJ+10] F. Deißenböck, B. Hummel, E. Jürgens, M. Pfähler, and B. Schätz. "Model Clone Detection in Practice". In: *Proc. of the Intl. Workshop on Software Clones (IWSC)*. ACM, 2010, pp. 57–64.

[Dij78] E. W. Dijkstra. *Tripreport E. W. Dijkstra, Marktoberdorf 24 July – 6 August 1978*. circulated privately. Aug. 1978. URL: http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD676.PDF.

[DK16] Á. Darvas and R. Konnerth. "System Architecture Recovery Based on Software Structure Model". In: *Proc. of the Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, 2016, pp. 109–114.

[DKB11] S. Duszynski, J. Knodel, and M. Becker. "Analyzing the Source Code of Multiple Software Variants for Reuse Potential". In: *Proc. of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2011, pp. 303–307.

[DKZ+12] T. Dhaliwal, F. Khomh, Y. Zou, and A. E. Hassan. "Recovering Commit Dependencies for Selective Code Integration in Software Product Lines". In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 202–211.

[DP09] S. Ducasse and D. Pollet. "Software Architecture Reconstruction: A Process-Oriented Taxonomy". In: *IEEE Transactions on Software Engineering (TSE)* 35.4 (2009), pp. 573–591.

[DPK+12] R. Damaševičius, P. Paškevičius, E. Karčiauskas, and R. Marcinkevičius. "Automatic Extraction of Features and Generation of Feature Models from Java Programs". In: *Information Technology and Control* 41.4 (2012), pp. 376–384.

[DRB+13] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. "An Exploratory Study of Cloning in Industrial Software Product Lines". In: *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2013, pp. 25–34.

[DRD99] S. Ducasse, M. Rieger, and S. Demeyer. "A Language Independent Approach for Detecting Duplicated Code". In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 1999, pp. 109–118.

[DRG+11] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. "Feature Location in Source Code: A Taxonomy and Survey". In: *Journal of Software: Evolution and Process* 25.1 (2011), pp. 53–95.

[DSP16] K. Damevski, D. Shepherd, and L. Pollock. "A Field Study of How Developers Locate Features in Source Code". In: *Empirical Software Engineering* 21.2 (Apr. 2016), pp. 724–747.

[EAA+08] M. Eaddy, A. V. Aho, G. Antoniol, and Y. G. Guéhéneuc. "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis". In: *Proc. of the Intl. Conference on Program Comprehension (ICPC)*. IEEE, 2008, pp. 53–62.

[EEM10] N. A. Ernst, S. M. Easterbrook, and J. Mylopoulos. "Code Forking in Open-Source Software: A Requirements Perspective". In: *Computing Research Repository (CoRR)* (2010).

[ELH+05] J. Estublier, D. Leblang, A. v. d. Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. "Impact of Software Engineering Research on the Practice of Software Configuration Management". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14.4 (Oct. 2005), pp. 383–430.

[Eve11] S. Even. *Graph Algorithms*. Second Edition. Cambridge University Press, 2011.

[FAH+15] J. Font, L. Arcega, Ø. Haugen, and C. Cetina. "Building Software Product Lines from Conceptualized Model Patterns". In: *Proc. of the Intl. Conference on Software Product Line (SPLC)*. ACM, 2015, pp. 46–55.

[FAH+16a] J. Font, L. Arcega, Ø. Haugen, and C. Cetina. "Feature Location in Model-Based Software Product Lines Through a Genetic Algorithm". In: *Proc. of the Intl. Conference on Software Reuse (ICSR)*. Vol. 9679. Lecture Notes in Computer Science. Springer, 2016, pp. 39–54.

[FAH+16b] J. Font, L. Arcega, Ø. Haugen, and C. Cetina. "Feature Location in Models Through a Genetic Algorithm Driven by Information Retrieval Techniques". In: *Proc. of the Intl. Conference on Model Driven Engineering Languages and Systems (MODELS)*. ACM, 2016, pp. 272–282.

[FAH+17] J. Font, L. Arcega, Ø. Haugen, and C. Cetina. "Achieving Feature Location in Families of Models through the use of Search-Based Software Engineering". In: *IEEE Transactions on Evolutionary Computation (TEVC)* (2017), pp. 363–377.

[FB14] N. Fenton and J. Bieman. *Software Metrics: A Rigorous and Practical Approach*. Third Edition. CRC Press, Inc., 2014.

[FBH+15] J. Font, M. Ballarín, Ø. Haugen, and C. Cetina. "Automating the Variability Formalization of a Model Family by Means of Common Variability Language". In: *Proc. of the Intl. Conference on Software Product Line (SPLC)*. ACM, 2015, pp. 411–418.

[FE99] H. Frank and J. Eder. "Towards an Automatic Integration of Statecharts". In: *Proc. of the Intl. Conference on Conceptual Modeling (ER)*. Vol. 1728. Lecture Notes in Computer Science. Springer, 1999, pp. 430–445.

[FKB+07] P. Frenzel, R. Koschke, A. P. J. Breu, and K. Angstmann. "Extending the Reflexion Method for Consolidating Software Variants into Product Lines". In: *Proc. of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2007, pp. 160–169.

[FLL+14] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. "Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants". In: *Proc. of the Intl. Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 391–400.

[FLL+15] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. "The ECCO Tool: Extraction and Composition for Clone-and-own". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 665–668.

[FMB+14] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. "Fine-grained and Accurate Source Code Differencing". In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 313–324.

[FMS+17] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, and G. Saake. "Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line". In: *Proc. of the Intl. Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2017, pp. 316–326.

[Fon17] J. Font Burdeus. "Location of Features as Model Fragments and their Co-Evolution". PhD Thesis. University of Oslo, Norway, 2017.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.3 (July 1987), pp. 319–349.

[FSD13] A. Ferrari, G. O. Spagnolo, and F. Dell'Orletta. "Mining Commonalities and Variabilities from Natural Language Documents". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 2013, pp. 116–120.

[FTS14] W. Fenske, T. Thüm, and G. Saake. "A Taxonomy of Software Product Line Reengineering". In: *Proc. of the Intl. Workshop on Variability Modeling in Software-Intensive Systems (VaMoS)*. ACM, 2014, 4:1–4:8.

[FV03] D. Faust and C. Verhoef. "Software Product Line Migration and Deployment". In: *Software: Practice and Experiences* 33.10 (2003), pp. 933–955.

[FWP+07] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction". In: *IEEE Transactions on Software Engineering (TSE)* 33.11 (2007), pp. 725–743.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[GM13] M. Goeminne and T. Mens. "A Comparison of Identity Merge Algorithms for Software Repositories". In: *Science of Computer Programming* 78.8 (2013), pp. 971–986.

[GS67] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine de Gruyter, 1967.

[Har87] D. Harel. "Statecharts: A Visual Formalism for Complex Systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231–274.

[HHK+13] A. Haber, K. Hölldobler, C. Kolassa, M. Look, B. Rumpe, K. Müller, and I. Schaefer. "Engineering Delta Modeling Languages". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 2013, pp. 22–31.

[HHK+15] A. Haber, K. Hölldobler, C. Kolassa, M. Look, K. Müller, B. Rumpe, I. Schaefer, and C. Schulze. "Systematic Synthesis of Delta Modeling Languages". In: *Software Tools for Technology Transfer* 17.5 (Oct. 2015), pp. 601–626.

[HKI08] Y. Higo, S. Kusumoto, and K. Inoue. "A Metric-based Approach to Identifying Refactoring Opportunities for Merging Code Clones in a Java Software System". In: *Journal of Software Maintenance and Evolution* 20.6 (Nov. 2008), pp. 435–461.

[HKM+13] A. Haber, C. Kolassa, P. Manhart, P. M. S. Nazari, B. Rumpe, and I. Schaefer. "First-Class Variability Modeling in Matlab/Simulink". In: *Proc. of the Intl. Workshop on Variability Modeling in Software-Intensive Systems (VaMoS)*. ACM, 2013, 4:1–4:8.

[HKR+11] A. Haber, T. Kutz, H. Rendel, B. Rumpe, and I. Schaefer. "Delta-oriented Architectural Variability Using MontiCore". In: *Proc. of the European Conference on Software Architecture (ECSA)*. ACM, 2011, 6:1–6:10.

[HKW08] F. Heidenreich, J. Kopcsek, and C. Wende. "FeatureMapper: Mapping Features to Models". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 943–944.

[HLE11] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed. "Reverse Engineering Feature Models from Programs' Feature Sets". In: *Proc. of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2011, pp. 308–312.

[HLE13] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed. "On Extracting Feature Models from Sets of Valid Feature Combinations". In: *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Vol. 7793. Lecture Notes in Computer Science. Springer, 2013, pp. 53–67.

[HM76] J. W. Hunt and M. D. McIlroy. *An Algorithm for Differential File Comparison*. 41. Computing Science Techical Report. Bell Laboratories, July 1976.

[HMO+08] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. "Adding Standardized Variability to Domain Specific Languages". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. IEEE, 2008, pp. 139–148.

[HMU06] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Third Edition. Addison-Wesley Longman Publishing, 2006.

[Hor90] S. Horwitz. "Identifying the Semantic and Textual Differences Between Two Versions of a Program". In: *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 1990, pp. 234–245.

[HRR12] A. Haber, J. O. Ringert, and B. Rumpe. *MontiArc: Architectural Modeling of Interactive Distributed and Cyber-Physical Systems*. AIB-2012-03. Techical Report. Rheinisch-Westfälische Technische Hochschule Aachen, Germany, Feb. 2012.

[HRW15] K. Hölldobler, B. Rumpe, and I. Weisemöller. "Systematically Deriving Domain-Specific Transformation Languages". In: *Proc. of the Intl. Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2015, pp. 136–145.

[HT73] J. Hopcroft and R. Tarjan. "Algorithm 447: Efficient Algorithms for Graph Manipulation". In: *Communications of the ACM* 16.6 (1973), pp. 372–378.

[HWL+14] S. Holthusen, D. Wille, C. Legat, S. Beddig, I. Schaefer, and B. Vogel-Heuser. "Family Model Mining for Function Block Diagrams in Automation Software". In: *Proc. of the Intl. Workshop on Reverse Variability Engineering (REVE)*. ACM, 2014, pp. 36–43.

[Int10] International Electrotechnical Commision (IEC). *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*. International Electrotechnical Commision (IEC), 2010.

[Int11] International Organization for Standardization (ISO). *ISO 26262: Road Vehicles – Functional Safety*. International Organization for Standardization (ISO), 2011.

[Int13] International Electrotechnical Commision (IEC). *IEC 61131-3: Programmable Logic Controllers – Part 3: Programming Languages*. International Electrotechnical Commision (IEC), Feb. 2013.

[Jac12] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[JB09] H. P. Jepsen and D. Beuche. "Running a Software Product Line: Standing Still is Going Backwards". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. Carnegie Mellon University, 2009, pp. 101–110.

[JBA+15] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki. "Maintaining Feature Traceability with Embedded Annotations". In: *Proc. of the Intl. Conference on Software Product Line (SPLC)*. ACM, 2015, pp. 61–70.

[JD88] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[JDB07] H. P. Jepsen, J. G. Dall, and D. Beuche. "Minimally Invasive Migration to Software Product Lines". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. IEEE, 2007, pp. 203–211.

[JK06] F. Jouault and I. Kurtev. "Transforming Models with ATL". In: *Proc. of the Intl. Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 3844. Lecture Notes in Computer Science. Springer, 2006, pp. 128–138.

[JL03] S. Jarzabek and S. Li. "Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique". In: *Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 2003, pp. 237–246.

[JL06] S. Jarzabek and S. Li. "Unifying Clones with a Generative Programming Technique: A Case Study: Practice Articles". In: *Journal of Software Maintenance and Evolution* 18.4 (July 2006), pp. 267–292.

[Joh93] J. H. Johnson. "Identifying Redundancy in Source Code Using Fingerprints". In: *Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM Press, 1993, pp. 171–183.

[Joh94] J. H. Johnson. "Substring Matching for Clone Detection and Change Tracking". In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 1994, pp. 120–126.

[KAK08] C. Kästner, S. Apel, and M. Kuhlemann. "Granularity in Software Product Lines". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 311–320.

[KAK09] C. Kästner, S. Apel, and M. Kuhlemann. "A Model of Refactoring Physically and Virtually Separated Features". In: *Proc. of the Intl. Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2009, pp. 157–166.

[KC99] R. Kazman and S. J. Carrière. "Playing Detective: Reconstructing Software Architecture from Available Evidence". In: *Automated Software Engineering* 6.2 (Apr. 1999), pp. 107–138.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. CMU/SEI-90-TR-021. Techical Report. Software Engineering Institute, Carnegie-Mellon University, USA, 1990.

[KDO11] C. Kästner, A. Dreiling, and K. Ostermann. *Variability Mining with LEADT*. Techical Report. Philipps Universitä Marburg, Germany, Jan. 2011.

[KDO14] C. Kästner, A. Dreiling, and K. Ostermann. "Variability Mining: Consistent Semi-automatic Detection of Product-Line Features". In: *IEEE Transactions on Software Engineering (TSE)* 40.1 (2014), pp. 67–82.

[KDP+09] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. "Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing". In: *Proc. of the Intl. Workshop on Workshop on Comparison and Versioning of Software Models (CVSM)*. IEEE, 2009, pp. 1–6.

[KFB+09] R. Koschke, P. Frenzel, A. P. J. Breu, and K. Angstmann. “Extending the Reflexion Method for Consolidating Software Variants into Product Lines”. In: *Software Quality Journal* 17.4 (Dec. 2009), pp. 331–366.

[KFF06] R. Koschke, R. Falke, and P. Frenzel. “Clone Detection Using Abstract Syntax Suffix Trees”. In: *Proc. of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2006, pp. 253–262.

[KG06] C. Kapser and M. W. Godfrey. “"Cloning Considered Harmful" Considered Harmful”. In: *Proc. of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2006, pp. 19–28.

[KH01] R. Komondoor and S. Horwitz. “Using Slicing to Identify Duplication in Source Code”. In: *Proc. of the Intl. Symposium on Static Analysis (SAS)*. Vol. 2126. Lecture Notes in Computer Science. Springer, 2001, pp. 40–56.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. “An Overview of AspectJ”. In: *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*. Vol. 2072. Lecture Notes in Computer Science. Springer, 2001, pp. 327–354.

[KHS+14] J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, and F. Damiani. “DeltaJ 1.5: Delta-oriented Programming for Java 1.5”. In: *Proc. of the Intl. Conference on Principles and Practices of Programming on the Java Platform (PPPJ)*. ACM, 2014, pp. 63–74.

[KK12] B. Klatt and M. Küster. “Respecting Component Architecture to Migrate Product Copies to a Software Product Line”. In: *Doctoral Symposium on Components and Architecture (WCOP)*. ACM, 2012, pp. 7–12.

[KK13] B. Klatt and M. Küster. “Improving Product Copy Consolidation by Architecture-aware Difference Analysis”. In: *Proc. of the Intl. Conference on Quality of Software Architectures (QoSA)*. ACM, 2013, pp. 117–122.

[KKI02] T. Kamiya, S. Kusumoto, and K. Inoue. “CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code”. In: *IEEE Transactions on Software Engineering (TSE)* 28.7 (2002), pp. 654–670.

[KKK07] K. Kim, H. Kim, and W. Kim. “Building Software Product Line from the Legacy Systems "Experience in the Digital Audio and Video Domain"”. In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. IEEE, 2007, pp. 171–180.

[KKK13] B. Klatt, M. Küster, and K. Krogmann. “A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies”. In: *Proc. of the Intl. Workshop on Reverse Variability Engineering (REVE)*. ACM, 2013, pp. 1–8.

[KKO+12] T. Kehrer, U. Kelter, M. Ohrndorf, and T. Sollbach. “Understanding Model Evolution through Semantically Lifting Model Differences with SiLift”. In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 638–641.

[KKP+12] T. Kehrer, U. Kelter, P. Pietsch, and M. Schmidt. “Adaptability of Model Comparison Tools”. In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. ACM, 2012, pp. 306–309.

[KKS14] B. Klatt, K. Krogmann, and C. Seidl. "Program Dependency Analysis for Consolidating Customized Product Copies". In: *Proc. of the Intl. Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 496–500.

[KKT11] T. Kehrer, U. Kelter, and G. Taentzer. "A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning". In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. IEEE, 2011, pp. 163–172.

[KKT13] T. Kehrer, U. Kelter, and G. Taentzer. "Consistency-Preserving Edit Scripts in Model Versioning". In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 191–201.

[KKW14] B. Klatt, K. Krogmann, and C. Wende. "Consolidating Customized Product Copies to Software Product Lines". In: *Proc. of the Workshop on Software-Reengineering & Evolution (WSRE)*. Gesellschaft für Informatik e.V. (GI), 2014, pp. 17–18.

[Kla14] B. Klatt. "Consolidation of Customized Product Copies into Software Product Lines". PhD Thesis. Karlsruhe Institute of Technology, Germany, 2014.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. "Aspect-Oriented Programming". In: *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*. Vol. 1241. Lecture Notes in Computer Science. Springer, 1997, pp. 220–242.

[KMP+05] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. "A Case Study in Refactoring a Legacy Component for Reuse in a Product Line". In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 2005, pp. 369–378.

[Kni02] J. C. Knight. "Safety Critical Systems: Challenges and Directions". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 2002, pp. 547–550.

[Kos07] R. Koschke. "Survey of Research on Software Clones". In: *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proceedings 06301. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[KPK+15] T. Kehrer, C. Pietsch, U. Kelter, D. Strüber, and S. Vaupel. "An Adaptable Tool Environment for High-Level Differencing of Textual Models". In: *Proc. of the Intl. Workshop on OCL and Textual Modeling*. Vol. 1512. CEUR-WS.org, 2015, pp. 62–72.

[KPP06a] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. "Merging Models with the Epsilon Merging Language (EML)". In: *Proc. of the Intl. Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 4199. Lecture Notes in Computer Science. Springer, 2006, pp. 215–229.

[KPP06b] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. "The Epsilon Object Language (EOL)". In: *Proc. of the European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*. Vol. 4066. Lecture Notes in Computer Science. Springer, 2006, pp. 128–142.

[KPP08] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. "The Epsilon Transformation Language". In: *Proc. of the Intl. Conference on Theory and Practice of Model Transformations (ICMT)*. Vol. 5063. Lecture Notes in Computer Science. Springer, 2008, pp. 46–60.

[KPR+11] D. S. Kolovos, R. F. Paige, L. M. Rose, and J. Williams. "Integrated Model Management with Epsilon". In: *Proc. of the European Conference on Modeling Foundations and Applications (ECMFA)*. Vol. 6698. Lecture Notes in Computer Science. Springer, 2011, pp. 391–392.

[Kri01] J. Krinke. "Identifying Similar Code with Program Dependence Graphs". In: *Proc. of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2001, pp. 301–309.

[Kru01] C. W. Krueger. "Easing the Transition to Software Mass Customization". In: *Proc. of the Intl. Workshop on Software Product-Family Engineering (PFE)*. Vol. 2290. Lecture Notes in Computer Science. Springer, 2001, pp. 282–293.

[Kru02] C. W. Krueger. "Practical Strategies and Techniques for Adopting Software Product Lines". In: *Proc. of the Intl. Conference on Software Reuse (ICSR)*. Vol. 2319. Lecture Notes in Computer Science. Springer, 2002, pp. 349–350.

[Kru08] C. W. Krueger. "The BigLever Software Gears Unified Software Product Line Engineering Framework". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. IEEE, 2008, pp. 353–353.

[KSV09] P. Klint, T. van der Storm, and J. Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation". In: *Proc. of the Working Conference on Source Code Manipulation and Analysis (SCAM)*. IEEE, 2009, pp. 168–177.

[KWN05] U. Kelter, J. Wehren, and J. Niere. "A Generic Difference Algorithm for UML Models". In: *Software Engineering*. Lecture Notes in Informatics 64 (2005), pp. 105–116.

[LAD+17] M. La Rosa, W. M. P. van der Aalst, M. Dumas, and F. P. Milani. "Business Process Variability Modeling: A Survey". In: *ACM Computing Surveys (CSUR)* 50.1 (Mar. 2017), 2:1–2:45.

[Lan64] P. J. Landin. "The Mechanical Evaluation of Expressions". In: *The Computer Journal* 6.4 (1964), pp. 308–320.

[LBC16] R. Lapeña, M. Ballarín, and C. Cetina. "Towards Clone-and-own Support: Locating Relevant Methods in Legacy Products". In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*. ACM, 2016, pp. 194–203.

[LBG17] L. Linsbauer, T. Berger, and P. Grünbacher. "A Classification of Variation Control Systems". In: *Proc. of the Intl. Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 2017, pp. 49–62.

[LBL+15] S. Lity, J. Bürdek, M. Lochau, M. Berens, A. Schürr, and I. Schaefer. "Re-Engineering Automation Systems as Dynamic Software Product Lines". In: *Tagungsband des Dagstuhl-Workshops Modellbasierte Entwicklung eingebetteter Systeme (MBEES)*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2015, pp. 43–52.

[LBL06] J. Liu, D. Batory, and C. Lengauer. "Feature Oriented Refactoring of Legacy Applications". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. ACM, 2006, pp. 112–121.

[LC13] M. A. Laguna and Y. Crespo. "A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring". In: *Science of Computer Programming* 78.8 (2013), pp. 1010–1034.

[LCC14] Z. Liang, Y. Cheng, and J. Chen. "A Novel Optimized Path-Based Algorithm for Model Clone Detection". In: *Journal of Software* 9.7 (2014), pp. 1810–1817.

[LCK+09] H. Lee, H. Choi, K. C. Kang, D. Kim, and Z. Lee. "Experience Report on Using a Domain Model-Based Extractive Approach to Software Product Line Asset Development". In: *Proc. of the Intl. Conference on Software Reuse (ICSR)*. Vol. 5791. Lecture Notes in Computer Science. Springer, 2009, pp. 137–149.

[LDU+13] M. La Rosa, M. Dumas, R. Uba, and R. Dijkman. "Business Process Model Merging: An Approach to Business Process Consolidation". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22.2 (Mar. 2013), 11:1–11:42.

[Lee61] C. Y. Lee. "An Algorithm for Path Connections and Its Applications". In: *IRE Transactions on Electronic Computers* EC-10.3 (Sept. 1961), pp. 346–365.

[Lev66] V. I. Levenshtein. "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals". In: *Soviet Physics Doklady* 10.8 (1966), pp. 707–710.

[LFL+15] L. Linsbauer, S. Fischer, R. E. Lopez-Herrejon, and A. Egyed. "Using Traceability for Incremental Construction and Evolution of Software Product Portfolios". In: *Proc. of the Intl. Symposium on Software and Systems Traceability (SST)*. IEEE, 2015, pp. 57–60.

[LFP+16] R. Lapeña, J. Font, F. Pérez, and C. Cetina. "Improving Feature Location by Transforming the Query from Natural Language into Requirements". In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*. ACM, 2016, pp. 362–369.

[LGB+12] R. E. Lopez-Herrejon, J. A. Galindo, D. Benavides, S. Segura, and A. Egyed. "Reverse Engineering Feature Models with Evolutionary Algorithms: An Exploratory Study". In: *Proc. of the Intl. Conference on Search Based Software Engineering (SSBSE)*. Vol. 7515. Lecture Notes in Computer Science. Springer, 2012, pp. 168–182.

[LGJ07] Y. Lin, J. Gray, and F. Jouault. "DSMDiff: a differentiation tool for domain-specific models". In: *European Journal of Information Systems* 16.4 (Aug. 2007), pp. 349–361.

[LLE17] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. "Variability Extraction and Modeling for Product Variants". In: *Software & Systems Modeling* (2017), pp. 1–21.

[LLG+15] R. E. Lopez-Herrejon, L. Linsbauer, J. A. Galindo, J. A. Parejo, D. Benavides, S. Segura, and A. Egyed. "An Assessment of Search-Based Techniques for Reverse Engineering Feature Models". In: *Journal of Systems and Software (JSS)* 103 (2015), pp. 353–369.

[LLL+13] S. Lity, R. Lachmann, M. Lochau, and I. Schaefer. *Delta-oriented Software Product Line Test Models – The Body Comfort System Case Study*. 2012-07. Technische Universität Braunschweig, Germany, 2013.

[LLM+04] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. "CP-Miner: A Tool for Finding Copy-Paste and Related Bugs in Operating System Code". In: *Proc. of the Symposium on Opearting Systems Design & Implementation (OSDI)*. USENIX Association, 2004, pp. 20–20.

[LLM+06] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code". In: *IEEE Transactions on Software Engineering (TSE)* 32.3 (2006), pp. 176–192.

[LMZ+06] H. Liu, Z. Ma, L. Zhang, and W. Shao. "Detecting Duplications in Sequence Diagrams Based on Suffix Trees". In: *Proc. of the Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2006, pp. 269–276.

[LRW11] C. Li, M. Reichert, and A. Wombacher. "Mining Business Process Variants: Challenges, Scenarios, Algorithms". In: *Data & Knowledge Engineering* 70.5 (2011), pp. 409–434.

[LSR05] N. Loughran, A. Sampaio, and A. Rashid. "From Requirements Documents to Feature Models for Aspect Oriented Product Line Implementation". In: *Proc. of the Intl. Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 3844. Lecture Notes in Computer Science. Springer, 2005, pp. 262–271.

[LSR07] F. J. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[LSS18] Y. Li, S. Schulze, and G. Saake. "Extracting Features from Requirements: Achieving Accuracy and Automation with Neural Networks". In: *Proc. of the Intl. Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2018, pp. 477–481.

[LT03] M. Lucas and D. Tilbury. "A Study of Current Logic Design Practices in the Automotive Manufacturing Industry". In: *International Journal of Human-Computer Studies* 59.5 (2003), pp. 725–753.

[Luc82] É. Lucas. *Récréations Mathématiques*. in French. Gauthier-Villars Paris, 1882, pp. 47–51.

[LXX+14] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao. "Detecting Differences Across Multiple Instances of Code Clones". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 164–174.

[LZR+17] Y. Li, C. Zhu, J. Rubin, and M. Chechik. "FHistorian: Locating Features in Version Histories". In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*. ACM, 2017, pp. 49–58.

[MA02] D. Muthig and C. Atkinson. "Model-Driven Product Line Architectures". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. Vol. 2379. Lecture Notes in Computer Science. Springer, 2002, pp. 110–129.

[Mal15] R. Malhotra. *Empirical Research in Software Engineering: Concepts, Analysis, and Applications*. Chapman & Hall/CRC, 2015.

[Mar16] J. Martínez. "Mining Software Artefact Variants for Product Line Migration and Analysis". PhD Thesis. University of Luxembourg, Luxembourg and Pierre and Marie Curie University, Paris, France, 2016.

[MCP+08] L. Murta, C. Corrêa, J. G. Prudêncio, and C. Werner. "Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System". In: *Proc. of the Intl. Workshop on Workshop on Comparison and Versioning of Software Models (CVSM)*. ACM, 2008, pp. 25–30.

[Men02] T. Mens. "A State-of-the-Art Survey on Software Merging". In: *IEEE Transactions on Software Engineering (TSE)* 28.5 (May 2002), pp. 449–462.

[MGC+16a] D. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin, and B. Baudry. "Puzzle: A Tool for Analyzing and Extracting Specification Clones in DSLs". In: *Proc. of the Intl. Conference on Software Reuse (ICSR)*. Vol. 9679. Lecture Notes in Computer Science. Springer, 2016, pp. 393–396.

[MGC+16b] D. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin, B. Baudry, and G. Le Guernic. "Reverse-Engineering Reusable Language Modules from Legacy Domain-Specific Languages". In: *Proc. of the Intl. Conference on Software Reuse (ICSR)*. Vol. 9679. Lecture Notes in Computer Science. Springer, 2016, pp. 368–383.

[MGH05] A. Mehra, J. Grundy, and J. Hosking. "A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design". In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. ACM, 2005, pp. 204–213.

[Mil95] G. A. Miller. "WordNet: A Lexical Database for English". In: *Communications of the ACM* 38.11 (1995), pp. 39–41.

[MLM96] J. Mayrand, C. Leblanc, and E. M. Merlo. "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics". In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 1996, pp. 244–253.

[MM85] W. Miller and E. W. Myers. "A File Comparison Program". In: 15.11 (Nov. 1985), pp. 1025–1040.

[MOD+07] L. Murta, H. Oliveira, C. Dantas, L. G. Lopes, and C. Werner. "Odyssey-SCM: An Integrated Software Configuration Management Infrastructure for UML Models". In: *Science of Computer Programming* 65.3 (2007), pp. 249–274.

[Moo59] E. F. Moore. "The Shortest Path Through a Maze". In: *Proc. of the Intl. on the Theory of Switching*. Harvard University Press, 1959, pp. 285–292.

[MRR11a] S. Maoz, J. O. Ringert, and B. Rumpe. "ADDiff: Semantic Differencing for Activity Diagrams". In: *Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 2011, pp. 179–189.

[MRR11b] S. Maoz, J. O. Ringert, and B. Rumpe. "CDDiff: Semantic Differencing for Class Diagrams". In: *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*. Vol. 6813. Lecture Notes in Computer Science. Springer, 2011, pp. 230–254.

[MRS08] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[MTS+17] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017.

[Mye86] E. W. Myers. "An O(ND) Difference Algorithm and Its Variations". In: *Algorithmica* 1.1 (Nov. 1986), pp. 251–266.

[MZB+15a] J. Martínez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon. "Automating the Extraction of Model-Based Software Product Lines from Model Variants". In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 396–406.

[MZB+15b] J. Martínez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon. "Bottom-up Adoption of Software Product Lines: A Generic and Extensible Approach". In: *Proc. of the Intl. Conference on Software Product Line (SPLC)*. ACM, 2015, pp. 101–110.

[MZB+16] J. Martínez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon. "Name Suggestions During Feature Identification: The Variclouds Approach". In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*. ACM, 2016, pp. 119–123.

[MZB+17] J. Martínez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon. "Bottom-Up Technologies for Reuse: Automated Extractive Adoption of Software Product Lines". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 67–70.

[MZK+14] J. Martínez, T. Ziadi, J. Klein, and Y. Le Traon. "Identifying and Visualising Commonality and Variability in Model Variants". In: *Proc. of the European Conference on Modeling Foundations and Applications (ECMFA)*. Vol. 8569. Lecture Notes in Computer Science. Springer, 2014, pp. 117–131.

[NBK+14] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. "Mining Configuration Constraints: Static Analyses and Empirical Results". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 140–151.

[Nej08] S. Nejati. "Behavioural Model Fusion". PhD Thesis. University of Toronto, Canada, 2008.

[NNP+09] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. "Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection". In: *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Vol. 5503. Lecture Notes in Computer Science. Springer, 2009, pp. 440–455.

[NSC+07] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. "Matching and Merging of Statecharts Specifications". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 2007, pp. 54–64.

[NSC+12] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. "Matching and Merging of Variant Feature Specifications". In: *IEEE Transactions on Software Engineering (TSE)* 38.6 (2012), pp. 1355–1375.

[NSG14] M. Nöbauer, N. Seyff, and I. Groher. "Inferring Variability from Customized Standard Software Products". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 2014, pp. 284–293.

[Obj15] Object Management Group (OMG). *Unified Modeling Language (UML)*. Object Management Group (OMG), May 2015. URL: `http://www.omg.org/spec/UML/2.5`.

[Obj16a] Object Management Group (OMG). *Meta Object Facility (MOF)*. Object Management Group (OMG), Nov. 2016. URL: `http://www.omg.org/spec/MOF/2.5.1`.

[Obj16b] Object Management Group (OMG). *MOF Query/View/Tranformation (QVT) Specification.* Object Management Group (OMG), June 2016. URL: http://www.omg.org/spec/QVT/1.3.

[OJ09] A. Olszak and B. N. Jørgensen. "Remodularizing Java Programs for Comprehension of Features". In: *Proc. of the Intl. Workshop on Feature-Oriented Software Development (FOSD).* ACM, 2009, pp. 19–26.

[OJ12] A. Olszak and B. N. Jørgensen. "Remodularizing Java Programs for Improved Locality of Feature Implementations in Source Code". In: *Science of Computer Programming* 77.3 (2012), pp. 131–151.

[OMW05] H. Oliveira, L. Murta, and C. Werner. "Odyssey-VCS: A Flexible Version Control System for UML Model Elements". In: *Proc. of the Intl. Workshop on Software Configuration Management (SCM).* ACM, 2005, pp. 1–16.

[OS06] T. Oda and M. Saeki. "Meta-Modeling Based Version Control System for Software Diagrams". In: *IEICE Transactions on Information and Systems* E89-D (Apr. 2006), pp. 1390–1402.

[OZL+11] S. Oster, M. Zink, M. Lochau, and M. Grechanik. "Pairwise Feature-interaction Testing for SPLs: Potentials and Limitations". In: *Proc. of the Intl. Software Product Line Conference (SPLC).* ACM, 2011, 6:1–6:8.

[Par76] D. L. Parnas. "On the Design and Development of Program Families". In: *IEEE Transactions on Software Engineering (TSE)* SE-2.1 (Mar. 1976), pp. 1–9.

[Pat14] M. Patel. *Programming in MATLAB.* Pearson Education, 2014.

[Pat87] M. Q. Patton. *How to Use Qualitative Methods in Evaluation.* SAGE Publications, 1987.

[PB03] R. A. Pottinger and P. A. Berstein. "Merging Models Based on Given Correspondences". In: *Proc. of the Intl. Conference on Very Large Databases (VLDB).* Elsevier, 2003, pp. 862–873.

[PBK+07] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner. "Software Engineering for Automotive Systems: A Roadmap". In: *Future of Software Engineering (FOSE).* IEEE, 2007, pp. 55–71.

[PBL05] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005.

[PCA+13] L. Passos, K. Czarnecki, S. Apel, A. Wasowski, C. Kästner, and J. Guo. "Feature-oriented Software Evolution". In: *Proc. of the Intl. Workshop on Variability Modeling in Software-Intensive Systems (VaMoS).* ACM, 2013, 17:1–17:8.

[PGG+03] M. Pinzger, H. Gall, J.-F. Girard, J. Knodel, C. Riva, W. Pasman, C. Broerse, and J. G. Wijnstra. "Architecture Recovery for Product Families". In: *Proc. of the Intl. Workshop on Software Product-Family Engineering (PFE).* Vol. 3014. Lecture Notes in Computer Science. Springer, 2003, pp. 332–351.

[PKK+15] C. Pietsch, T. Kehrer, U. Kelter, D. Reuling, and M. Ohrndorf. "SiPL – A Delta-Based Modeling Framework for Software Product Line Engineering". In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE).* IEEE, 2015, pp. 852–857.

[Plu06] A. R. Plummer. "Model-in-the-Loop Testing". In: *Proc. of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 220.3 (2006), pp. 183–199.

[PMP+15] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. "SPOON: A Library for Implementing Analyses and Transformations of Java Source Code". In: *Software: Practice and Experiences* 46.9 (2015), pp. 1155–1179.

[PNN+09] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. "Complete and Accurate Clone Detection in Graph-based Models". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 2009, pp. 276–286.

[PPM04] T. Pedersen, S. Patwardhan, and J. Michelizzi. "WordNet::Similarity: Measuring the Relatedness of Concepts". In: *Proc. of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NACCL HLT)*. Association for Computational Linguistics, 2004, pp. 38–41.

[PS83] H. Partsch and R. Steinbrüggen. "Program Transformation Systems". In: *ACM Computing Surveys (CSUR)* 15.3 (Sept. 1983), pp. 199–236.

[PTD+16] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wasowski, K. Czarnecki, P. Borba, and J. Guo. "Coevolution of Variability Models and Related Software Artifacts". In: *Empirical Software Engineering* 21.4 (Aug. 2016), pp. 1744–1793.

[PTS+16] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer. "Synchronizing Software Variants with VariantSync". In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*. ACM, 2016, pp. 329–332.

[PXT+13] X. Peng, Z. Xing, X. Tan, Y. Yu, and W. Zhao. "Improving Feature Location using Structural Similarity and Iterative Graph Mapping". In: *Journal of Systems and Software (JSS)* 86.3 (2013), pp. 664–676.

[Rad12] Radio Technical Commission for Aeronautics (RTCA). *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. Radio Technical Commission for Aeronautics (RTCA), 2012.

[RBS12] D. Rattan, R. Bhatia, and M. Singh. "Model Clone Detection based on Tree Comparison". In: *Proc. of the Annual India Conference (INDICON)*. IEEE, 2012, pp. 1041–1046.

[RC07] C. K. Roy and J. R. Cordy. *A Survey on Software Clone Detection Research*. 2007-541. Techical Report. School of Computing, Queen's University, Kingston, Ontario, Canada, Sept. 2007.

[RC08] C. K. Roy and J. R. Cordy. "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization". In: *Proc. of the Intl. Conference on Program Comprehension (ICPC)*. IEEE, 2008, pp. 172–181.

[RC12] J. Rubin and M. Chechik. "Combining Related Products into Product Lines". In: *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Vol. 7212. Lecture Notes in Computer Science. Springer, 2012, pp. 285–300.

[RC13a] J. Rubin and M. Chechik. "A Framework for Managing Cloned Product Variants". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1233–1236.

[RC13b] J. Rubin and M. Chechik. "Domain Engineering: Product Lines, Languages, and Conceptual Models". In: Springer, 2013. Chap. A Survey of Feature Location Techniques, pp. 29–58.

[RC13c] J. Rubin and M. Chechik. "N-way Model Merging". In: *Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013, pp. 301–311.

[RC13d] J. Rubin and M. Chechik. "Quality of Merge-Refactorings for Product Lines". In: *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Vol. 7793. Lecture Notes in Computer Science. Springer, 2013, pp. 83–98.

[RCC13] J. Rubin, K. Czarnecki, and M. Chechik. "Managing Cloned Variants: A Framework and Experience". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 2013, pp. 101–110.

[RCC15] J. Rubin, K. Czarnecki, and M. Chechik. "Cloned Product Variants: From Ad-Hoc to Managed Software Product Lines". In: *Software Tools for Technology Transfer* 17.5 (Oct. 2015), pp. 627–646.

[RCK09] C. K. Roy, J. R. Cordy, and R. Koschke. "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach". In: *Science of Computer Programming* 74.7 (May 2009), pp. 470–495.

[RH09] P. Runeson and M. Höst. "Guidelines for Conducting and Reporting Case Study Research in Software Engineering". In: *Empirical Software Engineering* 14.2 (Apr. 2009), pp. 131–164.

[Roc75] M. J. Rochkind. "The Source Code Control System". In: *IEEE Transactions on Software Engineering (TSE)* SE-1.4 (Dec. 1975), pp. 364–370.

[Roy09] C. K. Roy. "Detection and Analysis of Near-Miss Software Clones". In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 2009, pp. 447–450.

[RPK10] U. Ryssel, J. Ploennigs, and K. Kabitzsch. "Automatic Variation-Point Identification in Function-Block-Based Models". In: *Proc. of the Intl. Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2010, pp. 23–32.

[RPK11] U. Ryssel, J. Ploennigs, and K. Kabitzsch. "Extraction of Feature Models from Formal Contexts". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 2011, 4:1–4:8.

[RPK12a] U. Ryssel, J. Ploennigs, and K. Kabitzsch. "Automatic Library Migration for the Generation of Hardware-in-the-Loop Models". In: *Science of Computer Programming* 77.2 (2012), pp. 83–95.

[RPK12b] U. Ryssel, J. Ploennigs, and K. Kabitzsch. "Reasoning of Feature Models from Derived Features". In: *Proc. of the Intl. Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 2012, pp. 21–30.

[RRL+04] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. "Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases". In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 2004, pp. 188–197.

[RRS+16] J. Richenhagen, B. Rumpe, A. Schloßer, C. Schulze, K. Thissen, and M. von Wenckstern. "Test-driven Semantical Similarity Analysis for Software Product Line Extraction". In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*. ACM, 2016, pp. 174–183.

[RSA+15] E. J. Rapos, A. Stevenson, M. H. Alalfi, and J. R. Cordy. "SimNav: Simulink Navigation of Model Clone Classes". In: *Proc. of the Working Conference on Source Code Manipulation and Analysis (SCAM)*. IEEE, 2015, pp. 241–246.

[Rub14] J. Rubin. "Cloned Product Variants: From Ad-hoc to Well-managed Software Reuse". PhD Thesis. University of Toronto, Canada, 2014.

[RW11] B. Rumpe and I. Weisemöller. "A Domain Specific Transformation Language". In: *Proc. of the Intl. Workshop on Models and Evolution (ME)*. 2011.

[Rys14] U. Ryssel. "Automatische Generierung von feature-orientierten Produktlinien aus Varianten von funktionsblockorientierten Modellen". PhD Thesis, in German. Technischen Universität Dresden, Germany, 2014.

[RZW15] I. Reinhartz-Berger, A. Zamansky, and Y. Wand. "Taming Software Variability: Ontological Foundations of Variability Mechanisms". In: *Proc. of the Intl. Conference on Conceptual Modeling (ER)*. Vol. 9381. Lecture Notes in Computer Science. Springer, 2015, pp. 399–406.

[RZW16] I. Reinhartz-Berger, A. Zamansky, and Y. Wand. "An Ontological Approach for Identifying Software Variants: Specialization and Template Instantiation". In: *Proc. of the Intl. Conference on Conceptual Modeling (ER)*. Vol. 9974. Lecture Notes in Computer Science. Springer, 2016, pp. 98–112.

[Sab08] M. Sabetzadeh. "Merging and Consistency Checking of Distributed Models". PhD Thesis. University of Toronto, Canada, 2008.

[SBB+10] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. "Delta-Oriented Programming of Software Product Lines". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. Vol. 6287. Lecture Notes in Computer Science. Springer, 2010, pp. 77–91.

[SBD11] I. Schaefer, L. Bettini, and F. Damiani. "Compositional Type-checking for Delta-oriented Programming". In: *Proc. of the Intl. Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 2011, pp. 43–56.

[SC12] M. Stephan and J. R. Cordy. *A Survey of Methods and Applications of Model Comparison*. 2011-582. Version 3. Techical Report. School of Computing, Queen's University, Kingston, Ontario, Canada, June 2012.

[SC13] M. Stephan and J. R. Cordy. "A Survey of Model Comparison Approaches and Applications". In: *Proc. of the Intl. Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SCITEPRESS, 2013, pp. 265–277.

[Sch07] M. Schmidt. "Transformations of UML 2 Models Using Concrete Syntax Patterns". In: *Proc. of the Intl. Workshop on Rapid Integration of Software Engineering Techniques (RISE)*. Vol. 4401. Lecture Notes in Computer Science. Springer, 2007, pp. 130–143.

[SD10] I. Schaefer and F. Damiani. "Pure Delta-oriented Programming". In: *Proc. of the Intl. Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2010, pp. 49–56.

[SE03] M. Sabetzadeh and S. Easterbrook. "Analysis of Inconsistency in Graph-Based Viewpoints: A Category-Theoretical Approach". In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. IEEE, 2003, pp. 12–21.

[SE06] M. Sabetzadeh and S. Easterbrook. "View Merging in the Presence of Incompleteness and Inconsistency". In: *Requirements Engineering* 11.3 (June 2006), pp. 174–193.

[Sei15] C. Seidl. "Integrated Management of Variability in Space and Time in Software Families". PhD Thesis. Technische Universität Dresden, Germany, 2015.

[SGS02] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. "AspectC++: An Aspect-oriented Extension to the C++ Programming Language". In: *Proc. of the Intl. Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications (CRPIT)*. Australian Computer Society, 2002, pp. 53–60.

[SH04] M. Staples and D. Hill. "Experiences Adopting Software Product Line Development without a Product Line Architecture". In: *Proc. of the Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2004, pp. 176–183.

[SHA12] C. Seidl, F. Heidenreich, and U. Aßmann. "Co-Evolution of Models and Feature Mapping in Software Product Lines". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 2012, pp. 76–85.

[SHM+16] M. R. A. Setyautami, R. Hähnle, R. Muschevici, and A. Azurat. "A UML Profile for Delta-oriented Programming to Support Software Product Line Engineering". In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*. ACM, 2016, pp. 45–49.

[SLB+11] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. "Reverse Engineering Feature Models". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 461–470.

[SLF+08] P. Sánchez, N. Loughran, L. Fuentes, and A. Garcia. "Engineering Languages for Specifying Product-Derivation Processes in Software Product Lines". In: *Proc. of the Intl. Conference on Software Language Engineering (SLE)*. Vol. 5452. Lecture Notes in Computer Science. Springer, 2008, pp. 188–207.

[SNE+07] M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chechik. "A Relationship-Driven Framework for Model Merging". In: *Proc. of the Intl. Workshop on Modeling in Software Engineering (MISE)*. IEEE, 2007, pp. 2–7.

[SO01] C. Stoermer and L. O'Brien. "MAP - Mining Architectures for Product Line Evaluations". In: *Proc. of the Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, 2001, pp. 35–44.

[SRA+14] S. She, U. Ryssel, N. Andersen, A. Wasowski, and K. Czarnecki. "Efficient Synthesis of Feature Models". In: *Information and Software Technology* 56.9 (2014), pp. 1122–1143.

[SRA+16] D. Strüber, J. Rubin, T. Arendt, M. Chechik, G. Taentzer, and J. Plöger. "RuleMerger: Automatic Construction of Variability-Based Model Transformation Rules". In: *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Vol. 9633. Lecture Notes in Computer Science. Springer, 2016, pp. 122–140.

[SRC+12] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. "Software Diversity: State of the Art and Perspectives". In: *Software Tools for Technology Transfer* 14.5 (Oct. 2012), pp. 477–495.

[SRC+15] D. Strüber, J. Rubin, M. Chechik, and G. Taentzer. "A Variability-Based Approach to Reusable and Efficient Model Transformations". In: *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Vol. 9033. Lecture Notes in Computer Science. Springer, 2015, pp. 283–298.

[SRP10] T. Savage, M. Revelle, and D. Poshyvanyk. "FLAT3: Feature Location and Textual Tracing Tool". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 255–258.

[SRS13] S. Schulze, O. Richers, and I. Schaefer. "Refactoring Delta-oriented Software Product Lines". In: *Proc. of the Intl. Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 2013, pp. 73–84.

[SSA14] C. Seidl, I. Schaefer, and U. Aßmann. "DeltaEcore – A Model-Based Delta Language Generation Framework". In: *Modellierung*. Lecture Notes in Informatics. Gesellschaft für Informatik e.V. (GI), 2014, pp. 81–96.

[SSS15] A. Shatnawi, A. Seriai, and H. Sahraoui. "Recovering Architectural Variability of a Family of Product Variants". In: *Proc. of the Intl. Conference on Software Reuse (ICSR)*. Vol. 8919. Lecture Notes in Computer Science. Springer, 2015, pp. 17–33.

[SSS17] S. Schuster, C. Seidl, and I. Schaefer. "Towards a Development Process for Maturing Delta-oriented Software Product Lines". In: *Proc. of the Intl. Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2017, pp. 41–50.

[STK+12] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. "Variant-preserving Refactoring in Feature-oriented Software Product Lines". In: *Proc. of the Intl. Workshop on Variability Modeling in Software-Intensive Systems (VaMoS)*. ACM, 2012, pp. 73–81.

[Stö10] H. Störrle. "Towards Clone Detection in UML Domain Models". In: *Proc. of the European Conference on Software Architecture (ECSA)*. ACM, 2010, pp. 285–293.

[Stö13] H. Störrle. "Towards Clone Detection in UML Domain Models". In: *Software & Systems Modeling* 12.2 (May 2013), pp. 307–329.

[Stö15] H. Störrle. "Effective and Efficient Model Clone Detection". In: *Software, Services, and Systems*. Vol. 8950. Lecture Notes in Computer Science. Springer, 2015, pp. 440–457.

[SV06] T. Stahl and M. Völter. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, Inc., 2006.

[SW16] F. Schwägerl and B. Westfechtel. "SuperMod: Tool Support for Collaborative Filtered Model-Driven Software Product Line Engineering". In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 822–827.

[SW17a] F. Schwägerl and B. Westfechtel. "Maintaining Workspace Consistency in Filtered Editing of Dynamically Evolving Model-driven Software Product Lines". In: *Proc. of the Intl. Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SCITEPRESS, 2017, pp. 15–28.

[SW17b] F. Schwägerl and B. Westfechtel. "Perspectives on Combining Model-driven Engineering, Software Product Line Engineering, and Version Control". In: *Proc. of the Intl. Workshop on Variability Modeling in Software-Intensive Systems (VaMoS)*. ACM, 2017, pp. 76–83.

[SWC+17] A. Schlie, D. Wille, L. Cleophas, and I. Schaefer. "Clustering Variation Points in MATLAB/Simulink Models Using Reverse Signal Propagation Analysis". In: *Proc. of the Intl. Conference on Software Reuse (ICSR)*. Vol. 10221. Lecture Notes in Computer Science. Springer, 2017, pp. 77–94.

[SWS+17] A. Schlie, D. Wille, S. Schulze, L. Cleophas, and I. Schaefer. "Detecting Variability in MATLAB/Simulink Models: An Industry-Inspired Technique and its Evaluation". In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*. ACM, 2017, pp. 215–224.

[SWS19] C. Seidl, D. Wille, and I. Schaefer. "Software Reuse – From Cloned Variants to Managed Software Product Lines". In: *Book on Automotive Software Engineering*. to be published. Atlantis Press/Springer, 2019.

[Tar71] R. Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *Proc. of the Annual Symposium on Switching and Automata Theory (SWAT)*. IEEE, Oct. 1971, pp. 114–121.

[TBK09] T. Thüm, D. Batory, and C. Kästner. "Reasoning About Edits to Feature Models". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 2009, pp. 254–264.

[TBW+07] C. Treude, S. Berlik, S. Wenzel, and U. Kelter. "Difference Computation of Large Models". In: *Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 2007, pp. 295–304.

[Tic82] W. F. Tichy. "Design, Implementation, and Evaluation of a Revision Control System". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 1982, pp. 58–67.

[Tic85] W. F. Tichy. "RCS – A System for Version Control". In: *Software: Practice and Experiences* 15.7 (1985), pp. 637–654.

[TKB+14] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. "FeatureIDE: An Extensible Framework for Feature-Oriented Software Development". In: *Science of Computer Programming* 79 (2014), pp. 70–85.

[UC04] S. Uchitel and M. Chechik. "Merging Partial Behavioural Models". In: *Proc. of the Intl. Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2004, pp. 43–52.

[Vis01] E. Visser. "Stratego: A Language for Program Transformation Based on Rewriting Strategies". In: *Proc. of the Intl. Conference on Rewriting Techniques and Applications (RTA)*. Vol. 2051. Lecture Notes in Computer Science. Springer, 2001, pp. 357–361.

[WBC+18] D. Wille, Ö. Babur, L. Cleophas, C. Seidl, M. van den Brand, and I. Schaefer. "Improving Custom-Tailored Variability Mining Using Outlier and Cluster Detection". In: *Science of Computer Programming* 163 (2018), pp. 62–84.

[WCR09] N. Weston, R. Chitchyan, and A. Rashid. "A Framework for Constructing Semantically Composable Feature Models from Natural Language Requirements". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 2009, pp. 211–220.

[Wes91] B. Westfechtel. "Structure-oriented Merging of Revisions of Software Documents". In: *Proc. of the Intl. Workshop on Software Configuration Management (SCM)*. ACM, 1991, pp. 68–79.

[Wil14] D. Wille. "Family Mining of State Charts". Master's Thesis. Technische Universität Braunschweig, Germany, 2014.

[WM13] J. Weiland and P. Manhart. "A Classification of Modeling Variability in Simulink". In: *Proc. of the Intl. Workshop on Variability Modeling in Software-Intensive Systems (VaMoS)*. ACM, 2013, 7:1–7:8.

[WM95] R. W. Weeks and J. J. Moskwa. "Automotive Engine Modeling for Real-Time Control Using MATLAB/SIMULINK". In: *Society of Automotive Engineers (SAE) International Congress & Exposition*. Society of Automotive Engineers (SAE) International, Feb. 1995.

[WRS+17] D. Wille, T. Runge, C. Seidl, and S. Schulze. "Extractive Software Product Line Engineering Using Model-Based Delta Module Generation". In: *Proc. of the Intl. Workshop on Variability Modeling in Software-Intensive Systems (VaMoS)*. ACM, 2017, pp. 36–43.

[WRW07] N. Walkinshaw, M. Roper, and M. Wood. "Feature Location and Extraction using Landmarks and Barriers". In: *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 2007, pp. 54–63.

[WS00] J. Whittle and J. Schumann. "Generating Statechart Designs from Scenarios". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. ACM, 2000, pp. 314–323.

[WSS+16] D. Wille, S. Schulze, C. Seidl, and I. Schaefer. "Custom-Tailored Variability Mining for Block-Based Languages". In: *Proc. of the Intl. Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE, 2016, pp. 271–282.

[WSS16] D. Wille, S. Schulze, and I. Schaefer. "Variability Mining of State Charts". In: *Proc. of the Intl. Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2016, pp. 63–73.

[WSW+04] V. Wahler, D. Seipel, J. Wolff von Gudenberg, and G. Fischer. "Clone Detection in Source Code by Frequent Itemset Techniques". In: *Proc. of the Working Conference on Source Code Manipulation and Analysis (SCAM)*. IEEE, 2004, pp. 128–135.

[WTS+16] D. Wille, M. Tiede, S. Schulze, C. Seidl, and I. Schaefer. "Identifying Variability in Object-Oriented Code Using Model-Based Code Mining". In: *Proc. of the Intl. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Vol. 9953. Lecture Notes in Computer Science. Springer, 2016, pp. 547–562.

[WWP+16] K. Wehling, D. Wille, M. Pluchator, and I. Schaefer. "Towards Reducing the Complexity of Enterprise Architectures by Identifying Standard Variants Using Variability Mining". In: *Automobil Symposium Wildau*. Technische Hochschule Wildau, Germany, 2016, pp. 37–43.

[WWS+17a] K. Wehling, D. Wille, C. Seidl, and I. Schaefer. "Automated Recommendations for Reducing Unnecessary Variability of Technology Architectures". In: *Proc. of the Intl. Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2017, pp. 1–10.

[WWS+17b] K. Wehling, D. Wille, C. Seidl, and I. Schaefer. "Decision Support for Reducing Unnecessary IT Complexity of Application Architectures". In: *Proc. of the Intl. Workshop on decision Making in Software ARCHitecture (MARCH)*. IEEE, 2017, pp. 161–168.

[WWS+17c] D. Wille, K. Wehling, C. Seidl, M. Pluchator, and I. Schaefer. "Variability Mining of Technical Architectures". In: *Proc. of the Intl. Systems and Software Product Line Conference (SPLC)*. Vol. A. **HITACHI Young Best Paper Award – Research Track**. ACM, 2017, pp. 39–48.

[WWS+18] K. Wehling, D. Wille, C. Seidl, and I. Schaefer. "Reducing Variability of Technically Related Software Systems in Large-Scale IT Landscapes". In: *Proc. of the Intl. Conference on Computer Science and Software Engineering (CASCON)*. **IBM Center for Advanced Studies Best Paper Award**. IBM Corporation, 2018, pp. 224–235.

[XS05] Z. Xing and E. Stroulia. "UMLDiff: An Algorithm for Object-Oriented Design Differencing". In: *Proc. of the Intl. Conference on Automated Software Engineering (ASE)*. ACM, 2005, pp. 54–65.

[Yan91] W. Yang. "Identifying Syntactic Differences Between Two Programs". In: *Software: Practice and Experiences* 21.7 (1991), pp. 739–755.

[YZZ+12] L. Yi, W. Zhang, H. Zhao, Z. Jin, and H. Mei. "Mining Binary Constraints in the Construction of Feature Models". In: *Proc. of the Intl. Conference on Requirements Engineering (RE)*. IEEE, 2012, pp. 141–150.

[ZFS+12] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane. "Feature Identification from the Source Code of Product Variants". In: *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2012, pp. 417–422.

[Zha10] X. Zhang. "Synthesize Software Product Line". In: *Proc. of the Intl. Conference on Software Engineering (ICSE)*. Vol. 2. ACM, 2010, pp. 341–342.

[Zha14] X. Zhang. "Developing Model-Driven Software Product Lines". PhD Thesis. University of Oslo, Norway, 2014.

[ZHM11] X. Zhang, Ø. Haugen, and B. Møller-Pedersen. "Model Comparison to Synthesize a Model-Driven Software Product Line". In: *Proc. of the Intl. Software Product Line Conference (SPLC)*. IEEE, 2011, pp. 90–99.

[ZHM12] X. Zhang, Ø. Haugen, and B. Møller-Pedersen. "Augmenting Product Lines". In: *Proc. of the Asia-Pacific Software Engineering Conference (APSEC)*. Vol. 1. IEEE, 2012, pp. 766–771.

[ZHP+14] T. Ziadi, C. Henard, M. Papadakis, M. Ziane, and Y. Le Traon. "Towards a Language-independent Approach for Reverse-engineering of Software Product Lines". In: *Proc. of the ACM Symposium on Applied Computing (SAC)*. ACM, 2014, pp. 1064–1071.

[ZSS+09] S. Zschaler, P. Sánchez, J. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. Araújo, and U. Kulesza. "VML* – a Family of Languages for Variability Management in Software Product Lines". In: *Proc. of the Intl. Conference on Software Language Engineering (SLE)*. Vol. 5969. Lecture Notes in Computer Science. Springer, 2009, pp. 82–102.

Curriculum Vitae

Personal Information

Name **David Wille**

Birthday **28.04.1989 in Münster**

Education

08/1995 – 07/1999 **Elementary School Oyten**

08/1999 – 07/2001 **Elementary School Pestalozzistraße Oyten**

08/2001 – 07/2008 **Cato Bontjes van Beek-Gymnasium Achim (High School)**

Professional Training / University

10/2009 – 08/2012 **Bachelor of Science in Computer Science**,
Technische Universität Carolo-Wilhelmina Braunschweig

10/2012 – 12/2014 **Master of Science in Computer Science**, *Graduation with Honors,*
Technische Universität Carolo-Wilhelmina Braunschweig

Professional Experience

10/2012 – 07/2013
09/2013 – 12/2014 **Research Assistant**,
Institute of Software Engineering and Automotive Informatics,
Technische Universität Carolo-Wilhelmina Braunschweig

01/2015 – 11/2018 **Research Associate / PhD Student**,
Working Group Model-Based Software Engineering,
Institute of Software Engineering and Automotive Informatics,
Technische Universität Carolo-Wilhelmina Braunschweig

Awards

05/2014 **ZSB//Student Award**,
Student Counselling of
Technische Universität Carolo-Wilhelmina Braunschweig,
Awarded for particular good accomplishments at Bachelor's level

11/2015 **AutoVision Graduate Award**,
Carl-Friedrich-Gauß-Faculty of
Technische Universität Carolo-Wilhelmina Braunschweig,
Awarded for outstanding accomplishments at Master's level

09/2017 **HITACHI Young Best Paper Award – Research Track**,
Intl. Systems and Software Product Line Conference (SPLC),
Awarded for the paper "Variability Mining of Technical Architectures"

10/2018 **IBM Center for Advanced Studies Best Paper Award**,
Intl. Conference on Computer Science and Software Engineering (CASCON),
Awarded for the paper "Reducing Variability of Technically Related Software Systems in Large-Scale IT Landscapes"

Technische Universität Carolo-Wilhelmina Braunschweig
Institut für Softwaretechnik und Fahrzeuginformatik

Mühlenpfordtstr. 23
D-38106 Braunschweig